

AFIT/DS/ENG/99-07

Approximation and Optimization of an Auditory Model  
for Realization in VLSI Hardware

DISSERTATION  
Samuel L. SanGregory

AFIT/DS/ENG/99-07

19991210 042

Approved for public release; distribution unlimited

**DTIC QUALITY INSPECTED 2**

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense or the U.S. Government.

AFIT/DS/ENG/99-07

Approximation and Optimization of an Auditory Model  
for Realization in VLSI Hardware

DISSERTATION

Presented to the Faculty of the School of Engineering  
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the  
Requirements for the Degree of  
Doctor of Philosophy

Samuel L. SanGregory, M.S.

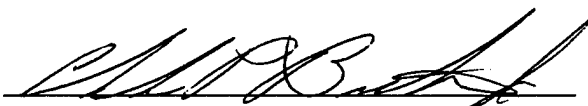
August 1999


Approved for public release; distribution unlimited

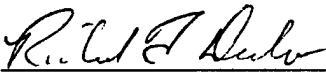
Approximation and Optimization of an Auditory Model  
for Realization in VLSI Hardware

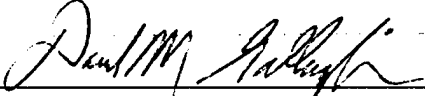
Samuel L. SanGregory, M.S.

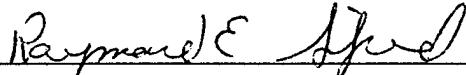
Approved:

  
Maj Charles P. Brothers, PhD (Chairman) Date 22 Nov 99

  
Dr. Timothy R. Anderson Date 18 Nov 99

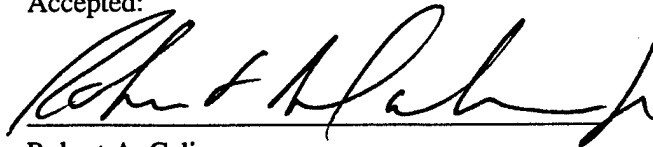
  
Dr. Richard F. Deckro (Dean's Representative) Date 22 Nov 99

  
Lt Col David M. Gallagher, PhD Date 18 Nov 99

  
Dr. Raymond E. Siferd Date 18 Nov 99

  
Dr. Tom S. Wailes Date 16 Nov 99

Accepted:



Robert A. Calico  
Dean, Graduate School of Engineering

## *Table of Contents*

	Page
List of Figures . . . . .	vii
List of Tables . . . . .	ix
Abstract . . . . .	x
 I. Introduction . . . . .	 1
1.1 Problem Statement . . . . .	2
1.2 Assumptions . . . . .	2
1.3 Objectives . . . . .	3
1.4 Summary . . . . .	4
 II. Background . . . . .	 5
2.1 Introduction . . . . .	5
2.2 Overview . . . . .	5
2.3 Middle Ear Filtering . . . . .	7
2.4 Spectral Analysis . . . . .	7
2.4.1 Gammatone Filtering . . . . .	7
2.4.2 Transmission Line Filtering . . . . .	7
2.4.3 Basilar Membrane Motion . . . . .	8
2.5 Neural Encoding . . . . .	9
2.5.1 AIM Functional Model . . . . .	10
2.5.2 AIM Physiological Model . . . . .	10
2.5.3 Neural Activity Pattern . . . . .	11
2.6 Time Interval Stabilization . . . . .	12
2.6.1 Strobed Temporal Integration . . . . .	12
2.6.2 Autocorrelation . . . . .	13
2.7 Summary . . . . .	14

	Page
III. Theory . . . . .	15
3.1 Introduction . . . . .	15
3.2 Middle Ear Filtering . . . . .	15
3.2.1 Implementation in AIM . . . . .	16
3.2.2 Approximation . . . . .	17
3.3 Spectral Analysis . . . . .	24
3.3.1 Implementation in AIM . . . . .	24
3.3.2 Approximation . . . . .	27
3.4 Neural Encoding–Rectification and Compression . . . . .	34
3.4.1 Implementation in AIM . . . . .	34
3.4.2 Approximation . . . . .	35
3.5 Neural Encoding–Adaptive Thresholding . . . . .	43
3.5.1 Implementation in AIM . . . . .	43
3.5.2 Approximation . . . . .	47
3.6 Integration Filtering . . . . .	54
3.6.1 Implementation in AIM . . . . .	54
3.6.2 Approximation . . . . .	55
3.7 Integrated AIM Approximation . . . . .	57
3.8 Summary . . . . .	57
IV. Hardware Implementation . . . . .	61
4.1 Introduction . . . . .	61
4.2 Filtering . . . . .	62
4.2.1 Algorithm . . . . .	62
4.2.2 Multipliers . . . . .	65
4.2.3 Memory . . . . .	66
4.2.4 Control . . . . .	69
4.3 Amplitude Compression . . . . .	74

	Page
4.3.1 Bit Shifter . . . . .	75
4.3.2 Logarithm Adjuster . . . . .	80
4.3.3 Conversion to Millibels . . . . .	80
4.4 Neural Encoding . . . . .	84
4.4.1 Adaptive Thresholding . . . . .	85
4.4.2 Integration Filter . . . . .	88
4.4.3 Control . . . . .	89
4.5 Summary . . . . .	91
V. Testing and Evaluation . . . . .	93
5.1 Introduction . . . . .	93
5.2 Functional Comparison . . . . .	93
5.3 Speed . . . . .	95
5.4 Size . . . . .	97
5.5 Power . . . . .	98
5.5.1 Multipliers . . . . .	100
5.5.2 Adders . . . . .	101
5.5.3 Latches . . . . .	101
5.5.4 State Machines . . . . .	102
5.5.5 Filterbank Power . . . . .	102
5.5.6 Logarithmic Compression . . . . .	103
5.5.7 Neural Encoding Processor . . . . .	103
5.5.8 Output Buffers and Clocking . . . . .	103
5.6 Summary . . . . .	104
VI. Conclusions and Recommendations . . . . .	106
6.1 Introduction . . . . .	106
6.2 Conclusions . . . . .	106

	Page
6.3 Recommendations for Further Research . . . . .	108
6.3.1 Layout Completion . . . . .	108
6.3.2 Integrated Circuit Interface . . . . .	108
6.3.3 Writable Coefficients Store . . . . .	109
6.3.4 Neural Encoding Approximations . . . . .	109
6.4 Summary . . . . .	110
Appendix A. Computation of All-Pole Gammatone Filter Constants . . . . .	111
Appendix B. State Output Equations for the Filterbank . . . . .	113
Appendix C. State Output Equations for NEP State Machine . . . . .	114
Appendix D. Rom Layout Generation Code . . . . .	115
Appendix E. VHDL Code . . . . .	118
Bibliography . . . . .	139



## *List of Figures*

Figure		Page
1.	AIM Block Diagram[1] . . . . .	6
2.	Simulated Basilar Membrane Motion for an Impulse . . . . .	8
3.	Simulated Neural Activity Pattern for an Impulse . . . . .	11
4.	Effect of Frequency Domain Thresholding (magnified 5X) . . . . .	12
5.	Stabilized Auditory Image for an Impulse . . . . .	13
6.	Anatomy of Middle Ear (after Lutman[2]) . . . . .	15
7.	Zwislocki's Functional Model and Circuit Equivalent (after [3]) . . . . .	16
8.	Giguere and Woodland's Outer/Middle Ear Model (after Giguere[4]) . . . . .	17
9.	AIM's Outer/Middle Ear Impulse Response . . . . .	18
10.	AIM's Outer/Middle Ear Frequency Response . . . . .	19
11.	Spice Simulation of Outer/Middle Ear Including the Canal . . . . .	20
12.	Spice Simulation of Outer/Middle Ear Without the Canal . . . . .	21
13.	Comparison Between AIM and the Approximate Middle Ear Filter . . . . .	21
14.	Impulse Response of 4 <sup>th</sup> Order 1 kHz Gammatone Filter . . . . .	25
15.	Comparison of Gammatone and APGF Impulse Response . . . . .	28
16.	Frequency Response of Gammatone and APGF . . . . .	29
17.	Gammatone and APGF Frequency Response with Middle Ear Filtering . . . . .	30
18.	Mitchell's Linear Approximation to $\log_2$ . . . . .	35
19.	Percent Error in Mitchell's Approximation . . . . .	36
20.	Comparison of Adjusted $\log_2$ Approximation and Actual $\log_2$ . . . . .	40
21.	Magnitude and Percent Errors of Optimized Approximation . . . . .	41
22.	Time Domain Adaptive Thresholding (after [5]) . . . . .	44
23.	Adaptive Thresholding Code from corti.c . . . . .	48
24.	Approximated Adaptive Thresholding . . . . .	53
25.	Difference Between AIM NAP and Approximated NAP . . . . .	53

Figure		Page
26.	Frequency Response of AIM's Integration Filter . . . . .	55
27.	Difference Between Exact and Approximated Integration Filters . . . . .	56
28.	NAP of "HAT" Generated by AIM . . . . .	58
29.	NAP of "HAT" Generated by Approximated Code . . . . .	58
30.	NAP of "1kHz Sinusoid" Generated by AIM . . . . .	59
31.	NAP of "1kHz Sinusoid" Generated by Approximated Code . . . . .	59
32.	Filtering Stage Block Diagram . . . . .	64
33.	Filtering Finite State Machine . . . . .	70
34.	Frequency Response of 12 Channel VHDL Filterbank . . . . .	74
35.	Frequency Response of 12 Channel AIM Filterbank . . . . .	75
36.	Bit Position Decode Logic . . . . .	76
37.	Complementary Bit Position Decode Logic . . . . .	77
38.	Barrel Shifter Array . . . . .	78
39.	Logarithm Adjusting Circuit . . . . .	81
40.	Optimized Adder Tree for Millibel Multiplication . . . . .	82
41.	Handshaking for Compression Stage . . . . .	83
42.	Block Diagram of Neural Encoding Processor (NEP) . . . . .	86
43.	State Machine for the Neural Encoding Processor . . . . .	89
44.	Neural Encoding Processor State Counter . . . . .	91
45.	Phoneme Recognition Rates for AIM and Approximation . . . . .	95
46.	Propagation Delay for Full-Adder Cells . . . . .	96

### *List of Tables*

Table		Page
1.	Comparison of Filter Run Times (seconds) . . . . .	22
2.	Comparison of Filter Bank Run Times (seconds) . . . . .	30
3.	Effect of Word Size on Filter Frequency . . . . .	33
4.	First Correction to Mitchell's Approximation . . . . .	38
5.	Second Correction to Mitchell's Approximation . . . . .	39
6.	VHDL Results for Approximation Adjustment Strategies . . . . .	40
7.	Lateral Leakage Parameters . . . . .	46
8.	Filter Section State Table . . . . .	71
9.	Filter State Jump Addresses . . . . .	72
10.	Neural Encoding Processor State Table . . . . .	90
11.	Neural Encoding Processor Jump States . . . . .	91
12.	Area Estimates For Architecture ( $\lambda$ ) . . . . .	99
13.	Spice Simulation Results for 6-bit Multiplier . . . . .	100
14.	Power Analysis for Neural Encoding Processor . . . . .	103

### *Abstract*

The Auditory Image Model (AIM) is a software tool set developed to functionally model the role of the ear in the human hearing process. AIM includes detailed filter equations for the major functional portions of the ear. Currently, AIM is run on a workstation and requires 10 to 100 times real-time to process audio information and produce an auditory image.

An all-digital approximation of the AIM which is suitable for implementation in very large scale integrated circuits is presented. This document details the mathematical models of AIM and the approximations and optimizations used to simplify the filtering and signal processing accomplished by AIM. Included are the details of an efficient multi-rate architecture designed for sub-micron VLSI technology to carry out the approximated equations. Finally, simulation results which indicate that the architecture, when implemented in  $0.8\mu\text{m}$  CMOS VLSI, will sustain real-time operation on a 32 channel system are included. The same tests also indicate that the chip will be approximately  $3.3\text{ mm}^2$ , and consume approximately 18 mW.

The details of a new and efficient method for computing an approximate logarithm (base two) on binary integers is also presented. The approximate logarithm algorithm is used to convert sound energy into millibels quickly and with low power. Additionally, the algorithm, is easily extended to compute an approximate logarithm in base ten which broadens the class of problems to which it may be applied.

# Approximation and Optimization of an Auditory Model for Realization in VLSI Hardware

## *I. Introduction*

*The hearing ear, and the seeing eye, the Lord hath made even both of them.*

Proverbs 20:12

There exists today a myriad of electronic devices which can gather, store, modify, and replay visual and auditory data. As of yet, however, man-made technology has been unable to replicate the Creator's unique design of the human auditory system. The task of reproducing auditory images through electronic means is hampered by our inability to fully grasp the intricate processing taking place within the sensory organs and the brain. The brain is a massively parallel data processing system; even if we could comprehend the algorithms necessary to program it, the hardware required to replicate the brain would be enormous. Still, we continue in our attempts to understand the complex processes by which sound waves are transformed into auditory images.

This research focuses on one very specific aspect of the auditory process: modeling the way in which the ear pre-processes auditory information for the brain. Previous research has established that we perceive sounds as auditory images. A computer program called the Auditory Image Model (AIM) produced by researchers in England generates computer approximations of the auditory image for sampled audio[1]. The objective of this research is to show that the existing computer model can be mapped efficiently into an application-specific integrated circuit.

The underlying thesis, which will enable the mapping of the algorithm into hardware, is that the complex algorithms currently used in auditory research may be simplified without significantly impairing the usefulness of the model. The existing algorithms were designed to closely match the physical characteristics of the ear. While they succeed in terms of matching the ear, these algorithms are also very mathematically intensive.

The approach chosen to speed up the modeling of the ear was to first develop new algorithms which approximate both the time and frequency domain response of the existing algorithms.

Throughout the design of these new algorithms, emphasis was placed on the efficiency with which they could be implemented into application specific VLSI hardware. The new algorithms were also tested in a set of phoneme recognition experiments and the results compared to results obtained using the original algorithms.

After a satisfactory set of algorithms was developed, a computer architecture was designed to execute them. The architecture was then modeled to verify it as a correct implementation of the approximated algorithms. The design of the architecture included characterizing the major components in order to obtain estimates on the actual power required and operating speed.

### *1.1 Problem Statement*

AIM has proven to be a very useful tool for auditory research, spanning topics from speech recognition to physiological studies of hearing. However, AIM is very complex and time consuming, requiring as much as 10 times real time to process sound on a typical workstation computer[16]. A specialized hardware accelerator is needed that can relieve the host CPU from the heavy computational load imposed by AIM. Such hardware will have significant impact on the utility of the model by allowing more and larger data sets to be processed. Additionally, a hardware accelerator would allow the real-time processing of data streams.

A direct implementation of AIM into hardware would be expensive in terms of VLSI area and power. Additionally, unless heavily pipelined, it would not achieve real-time operation. Therefore, before AIM can be implemented into hardware, a new set of algorithms must be developed which closely approximate the algorithms of AIM but require less processing power.

### *1.2 Assumptions*

Several assumptions were made early in the research to simplify and direct the effort. First, it was assumed that the functional algorithms of AIM sufficiently model the human auditory system. This assumption did not preclude the implementation of one of the physiological portions of AIM if it proved to have a more efficient mapping into hardware. Further, it was assumed that any implementation of the algorithms would be realized in an application specific integrated circuit (ASIC). In addition, it was assumed that the system (which may be more than one ASIC) would need to

communicate with standard workstation computers, requiring the consideration of an appropriate interface.

Initially, no assumptions were made as to how many filter channels are necessary to provide accurate auditory modeling, nor at what frequencies these filters should operate.<sup>1</sup> These parameters are left generic so that the end system may be configured according to user needs. It will be initially assumed that the system must operate at a data rate of 20 k-samples per second; however, the system will maintain enough flexibility to either support multiple sampling rates, or be re-programmable. In order to maintain as much flexibility as possible, it is assumed that the user of the system may want to bypass any combination of the model's stages, therefore bypass capability will also be considered.

### *1.3 Objectives*

The specific objectives this dissertation are:

1. To optimize or replace modules in the functional model of AIM so that the resulting code will map more efficiently into CMOS VLSI hardware.
2. To design and test using the VHSIC Hardware Description Language (VHDL) an architecture that will be able to perform the functional version of AIM faster than the C code while reducing the electrical power consumed. The goal is to approach real time in the data stream.
3. To layout and fabricate, as a minimum, one channel that will perform the functions of AIM from the ear opening to the neural activity pattern (NAP).<sup>2</sup>
4. To collect data from the prototype chip and project the full system power and speed from the data.
5. To complete the design of the full-scale system.

---

<sup>1</sup>Some researchers have used AIM as a front end for phoneme recognition with as few as 18 channels[6], while others claim much greater numbers of channels are required for accurate modeling of the hearing processes[7].

<sup>2</sup>This objective was later eliminated when it became apparent that the only difference between a single channel system and a fully functional multi-channel system would be the amount of on-chip memory.

#### *1.4 Summary*

This chapter established the motive and presented the overall objectives for the research, specifically to design a hardware accelerator for AIM. The path to the hardware solution includes characterizing, optimizing, and replacing most of the processing modules of AIM with new and simpler algorithms. Following the algorithm development a processing architecture is designed and tested.

This document is a record of the research effort beginning with Chapter 2 which reviews the present state of the art of auditory research. After the background review, the theory behind both current models of the human ear and the newly designed approximate model is discussed in Chapter 3 followed by the design of an application-specific processor in Chapter 4. Next, Chapter 5 summarizes the test results and provides an evaluation of operational parameters for the processor. Finally, Chapter 6 concludes with summary remarks and recommendations for further work.



## *II. Background*

### *2.1 Introduction*

In order to establish a foundation, this chapter provides a look at the current state of auditory research. Because of its importance to the auditory modeling community, most of the chapter is devoted to a detailed discussion of one auditory model, the Auditory Image Model (AIM). The discussion of AIM in this chapter is broken down into sections which follows the progression of information through the model.

### *2.2 Overview*

Researchers investigating topics ranging from music perception to speech recognition have for many years been studying the processes of the human ear. Out of the previous research, several hardware implementations of auditory models have been developed. For example, Lazzaro and Wawrzynek developed a system to transform audio data into what they referred to as a specialized representation of sound[8]. Similarly, Lyon developed an analog correlation chip using charge coupled device (CCD) technology which takes an analog input and produces an analog output correlogram in real time[9].

While the existing hardware implementations have been shown to be useful under specific conditions, many research applications require a more generalized system. The Auditory Image Model (AIM), is a physiologically-based computer program that has such flexibility[10, 1]. AIM was developed by researchers at the Medical Research Council, Applied Psychology Unit in Cambridge England. AIM, a software suite written in the C programming language in the late 1980's, was first released to researchers worldwide in the early 1990's. AIM version 8.1, released in September 1996, was used throughout this research.

While other methods exist for processing audio information (for example: windowed Fourier transform techniques), none have the physiological basis that AIM has. AIM was selected because of how it preserves time-domain information. Recent research indicates that the brain uses phase information to localize auditory sources as well as to distinguish desired sounds from background noise[7]. Because AIM preserves the time-domain properties of the sound it may have a greater potential for further auditory research.

AIM is a modular program which processes the data in sequential stages. These stages correspond to physical processes in the human auditory system. The organization of the program makes it easy to single out particular portions of the auditory system for detailed study. Additionally, the organization allows for easy replacement of functional blocks with different models. For example, the user may, through a command-line option, choose one of three models for the cochlear filtering.

AIM also supports two different methods of analysis: a functional method, and a physiological method. Figure 1 illustrates how the AIM software is constructed. Included in Figure 1 are the section numbers within this chapter where each stage is discussed.

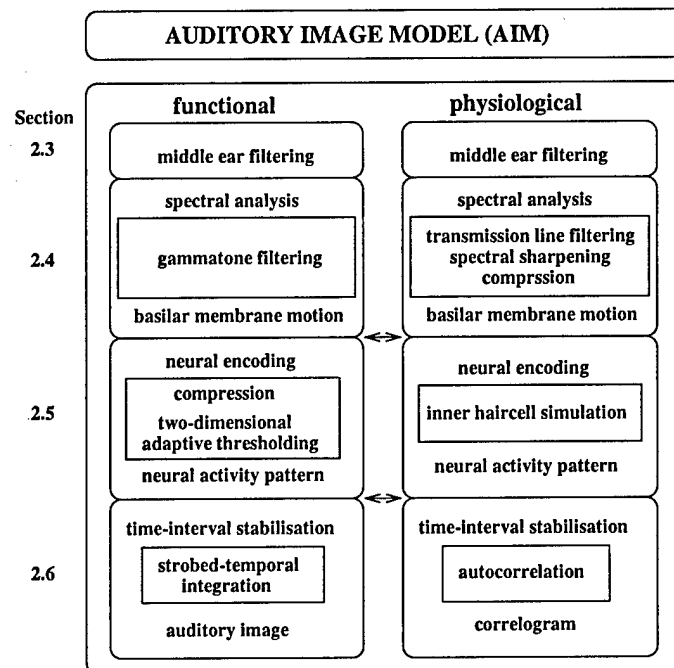


Figure 1: AIM Block Diagram[1]

The primary differences between the two sides of the AIM model rest in how the modeling is accomplished. On the physiological side, much effort went into developing models that accurately reflect the acousto-mechanical properties of the ear. On the functional side, however, the models are a systems approach where the transfer function is modeled rather than the mechanics. As the arrows between the sides indicate, a researcher may choose to use a combination of models from both sides of AIM by simply using command-line options.

### 2.3 Middle Ear Filtering

In breaking down the model further, the physiological and functional sides are divided into four major areas: middle ear filtering, spectral analysis, neural encoding, and time-interval stabilization. Both sides of AIM use the same middle-ear filtering mechanism, a bandpass filter centered near 1 kHz. The response of the filter rolls off at approximately 20 dB/decade on each side of 1 kHz.

Currently the middle ear filter is implemented in a more physiological approach through the use of a wave digital filter (WDF) developed by Christian Giguere[11]. The WDF model includes the external ear effects of the head and upper torso, but does not include the effects of the external ear flange[4]. Additionally, the model assumes the sound to be arriving at the head perpendicular to the ear.

### 2.4 Spectral Analysis

The second stage of AIM models the basilar membrane located in the inner ear within the cochlea (See Figure 1). As sound travels through the ear it is transformed into a pressure wave that, in the fluid filled cochlea, causes the basilar membrane to vibrate. Higher frequency waves cause the front section of the basilar membrane (closest to the eardrum) to move, while the low frequencies penetrate deeper causing motion toward the back of the basilar membrane. In the actual ear, the distribution of energy is continuous from the high frequencies to the low[12]. The discussion of the spectral analysis is divided here into first, the functional and second, the physiological model of the process. Finally, the format of the output from these models is presented.

**2.4.1 Gammatone Filtering.** In the functional version of AIM, the single stream of audio data which enters the model is separated into  $N$  different frequency bands through the use of passband filters with varying center frequencies. The number  $N$  is specified by the user through command-line options. The discussion of the derivation of the filter function can be found in [12, 13, 14, 15, 16, 17], while the filter bandwidth and spacing is discussed in[17, 18, 19, 20, 21].

**2.4.2 Transmission Line Filtering.** Beginning in the spectral analysis stage, the functional and physiological sides of AIM (See Figure 1) begin to differ. As with the functional model

just discussed, in the physiological mode, AIM again generates  $N$  different frequency bands. However, in the case of the physiological model, the filtering mechanism is not the the gammatone filter bank of the functional model. For the physiological case, the filtering is accomplished using a wave-digital filter (WDF) representation of a transmission-line model of the inner ear. Like the middle ear filter previously discussed, the WDF model attempts to capture the acousto-mechanical properties of the inner ear[11].

**2.4.3 Basilar Membrane Motion.** Regardless of the model used, the output from the second stage is referred to as the basilar membrane motion (BMM)[1]. The BMM essentially represents the position of discrete locations on the basilar membrane relative to its at-rest (quiet) position. An example of BMM for an impulse is seen in Figure 2.

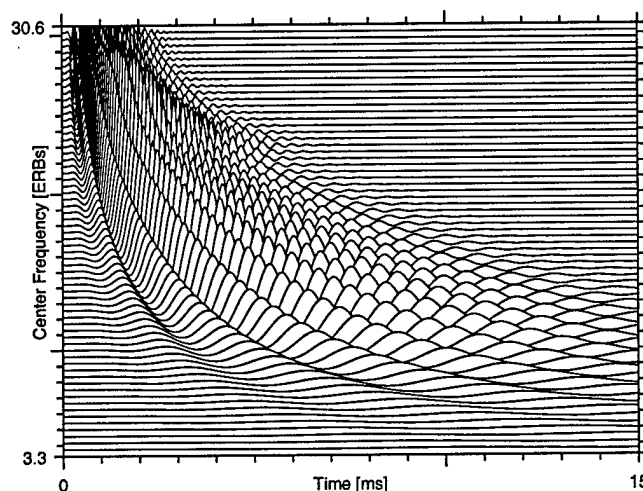


Figure 2: Simulated Basilar Membrane Motion for an Impulse

In Figure 2 the horizontal axis represents time in milliseconds starting with the onset of the sound data. The vertical axis is the equivalent rectangular bandwidth scale (ERB). As Slaney states, "The ERB is a psychoacoustic measure of the width of the auditory filter at each point along the cochlea... an ERB filter models the signal that is present within a single auditory nerve cell or channel."[17:p.2]. Patterson[1] credits the ERB filter to Glasberg and Moore[20] which he

further references to physiological research of Greenwood[21] as well psychoacoustic research of Patterson and Moore[22].

While several definitions for the ERB exist, AIM uses the more recent definition for the ERB which is presently accepted as more accurate[5]. AIM defines the ERB as:

$$ERB = 24.7(4.37f_o/1000 + 1) \quad (1)$$

where  $f_o$  represents the center frequency of the auditory filter. The ERB scale, which appears on the Y-axis of the figures generated by AIM is computed by integrating the inverse of the ERB function. The ERB scale was proposed by Moore[18] as an “instructive and convenient” method to plot psychoacoustical data. Equation 2 may be used to convert the ERB scale (S) back to filter center frequency (in Hertz).

$$f_o = (e^{0.10794S} - 1)/(4.37 \times 10^{-3}) \quad (2)$$

Unless otherwise specified, in the examples shown in this document, this scale is equivalent to a frequency span from 100 Hz to 6 kHz where 100 Hz is at the bottom of the figure. The filter spacing is not linear, but rather is selected so that there is equal overlap in the skirts of the filter spectra of adjacent filters.

## 2.5 Neural Encoding

The third stage in the AIM model is the transformation of the mechanical movement of the basilar membrane into electrical impulses which are sent to the brain via the auditory nerve. The transformation is accomplished by the inner hair cells which are located along the edge of the basilar membrane[10]. As these cells are bent under the pressure of the moving basilar membrane, they transmit electrical signals to the auditory nerve. These nerves only fire when compressed by the basilar membrane, not when they are returning to their resting position. Therefore, they essentially perform a half-wave rectification on the BMM.

In addition, these inner hair cells respond very quickly, and in a nonlinear manner, by adapting to the intensity level of the sound. Therefore, sounds with high intensity lessen the ability of

the ear to detect softer sounds. Finally, although the motion of the basilar membrane is frequency-dependent along its length, the inner hair cells tend to interact with neighbors, causing some smearing of the frequency separation.

**2.5.1 AIM Functional Model.** To model the behavior of the inner hair cells, the functional model of AIM performs several operations. First, the BMM is half-wave rectified to account for the fact that the inner hair cells only fire when compressed. Second, either a logarithm base 10, or an exponent of  $x^y$  ( $0 < y < 1$ ) of the rectified BMM is calculated on a point-by-point basis to model the compression in amplitude that takes place as the hair cells adapt to louder sounds. The choice regarding which form of compression is used remains a user option. Traditionally, the logarithmic form has been used; however, recent data has been published indicating that a  $X^{0.5}$  is also an effective model[23], particularly when using the physiological hair cell model.

After the rectification and compression is accomplished, the data is run through a process referred to as *two-dimensional adaptive thresholding*. The adaptive thresholding attempts to “reintroduce, and perhaps enhance, the contrast of features that appear in the basilar membrane motion[10:p.4:4].” As the name *two-dimensional adaptive thresholding* implies, the thresholding process is accomplished in both time and frequency.

In the time domain, adaptive thresholding is accomplished by comparing the current BMM value to the expected impulse response of the filter used to produce the BMM. If the current value of the BMM exceeds the expected value, then the simulated nerve fires, and the expected value is updated. The thresholding process prevents the natural ringing of the filters from causing multiple false firings of the inner hair cells.

In the frequency domain, the BMM is examined across filter neighbors to model the interaction between adjacent inner hair cells. The inter-filter effect depends on the filter frequency and spacing; the higher the center frequency and the further apart the filters, the less effect the filters have on one another. In any case, the effect of the frequency adaption is very small. The effect of frequency adaption is illustrated in Section 2.5.3.

**2.5.2 AIM Physiological Model.** The physiological hair cell model, as with all of the physiological algorithms, attempts to model as closely as possible the behavior of the inner hair

cells from the physiological perspective. To do so, AIM simulates a single hair cell for each filter frequency in the BMM. Like the physiological cochlear model, the hair cell is modeled using a wave digital filter. The present model was developed by Giguere with the theory behind the hair cell model being credited to Meddis[24].

**2.5.3 Neural Activity Pattern.** As Figure 1 illustrates, the output of the third stage of the model is referred to as a Neural Activity Pattern (NAP). The NAP represents the neural firings associated with each of the frequency channels generated in the BMM. Figure 3 illustrates the NAP for the same impulse that generated the BMM of Figure 2. Again, the horizontal axis is time, while the vertical axis represents the ERB spacings described in Section 2.4.3.

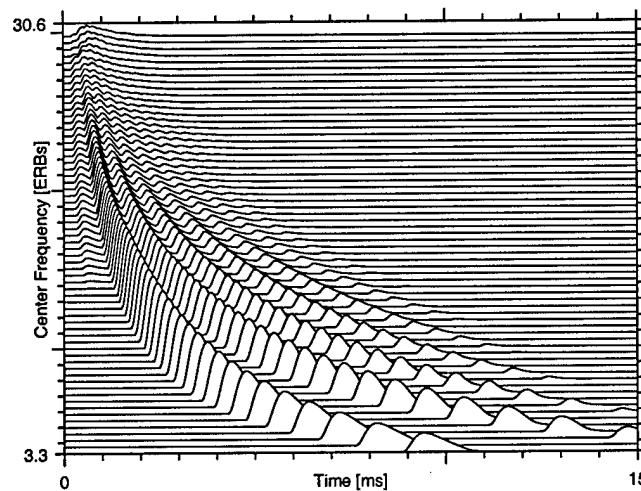


Figure 3: Simulated Neural Activity Pattern for an Impulse

Most of the features of the NAP are generated by the time-domain thresholding. To illustrate, observe Figure 4 which was generated by subtracting a NAP with the frequency adaption disabled from the NAP of the same stimulus with the frequency adaption enabled. In order to make the differences more pronounced, the features of Figure 4 have been magnified five times.

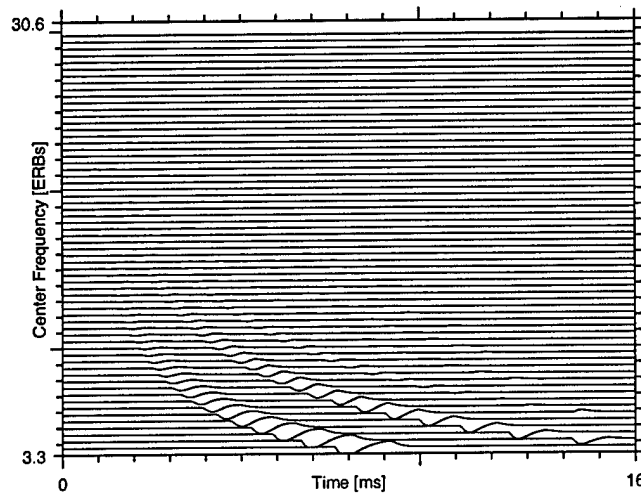


Figure 4: Effect of Frequency Domain Thresholding (magnified 5X)

## 2.6 Time Interval Stabilization

In the body, after the inner ear nerves fire, the electrical activity is transmitted to the brain via the auditory nerve. When the brain receives the neural activity, an auditory image is constructed and analyzed, resulting in the perception of the sounds. In AIM, the generation of the auditory image is simulated by AIM's fourth and final stage.

As seen in Figure 1, the final stage is titled time interval stabilization. For both the physiological and functional models, the auditory image is constructed in the final stage. The output of the time interval stabilization stage is referred to as the stabilized auditory image (SAI). As noted by Patterson, the intent of the SAI is to "preserve what we hear in the sound and remove what we do not hear"[10:p.5:1].

**2.6.1 Strobed Temporal Integration.** The generation of the SAI in the functional model is accomplished by a process termed strobed temporal integration, while in the physiological model it is done by autocorrelation. Thus in the physiological model the output is more correctly referred to as a correlogram rather than a stabilized auditory image.

In the functional case, each channel of the NAP is buffered and stored with a linear decay in magnitude of 2.5% per ms. Each buffer channel is continuously scanned for local maxima as it



is filled. When a local maxima occurs, an integrator for that channel is strobed (or fires), which adds the buffer contents to the corresponding channel in the auditory image on a point-by-point basis, with the local maxima stored in time at the 0 ms location in the image. The time axis on the image is thus a record of the integrated channels up to the current integration-initiating event[1]. An example of an auditory image is shown in Figure 5. Figure 5 differs from the BMM and NAP figures shown previously (Figures 2 and 3) in that it was generated from a repeated impulse stream rather than a single impulse.

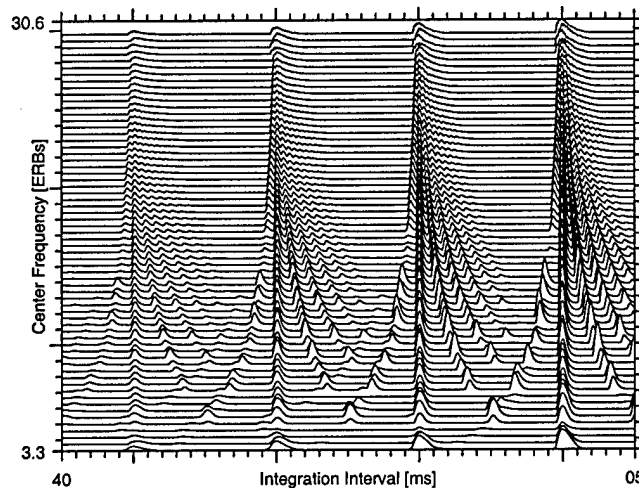


Figure 5: Stabilized Auditory Image for an Impulse

The auditory image shown in Figure 5 is actually the last frame of several. An image similar to Figure 5 is produced for each local maximum that occurs in the NAP. For periodic signals (such as the pulse train shown in Figure 5) the SAI becomes static; that is, all image frames are nearly the same. For aperiodic sounds (such as speech) each frame is different. By viewing the image frames in succession, we can visualize the changing nature of the auditory image.

**2.6.2 Autocorrelation.** An alternative to the strobed temporal integration is the autocorrelation, provided by the physiological side of AIM, where a recursive or running autocorrelation algorithm is implemented. The output is very similar to the SAI produced by the strobed temporal integration, however, it is more symmetric and has larger level contrasts. Interestingly, Patterson

points out that there is currently no physiological evidence that the ear performs autocorrelation, yet the algorithm is applied to the physiological side of the model[1]. Additionally, AIM also provides a fast Fourier transform (FFT) based algorithm which is functionally equivalent to the autocorrelation algorithm.

## 2.7 *Summary*

Of the different methods of simulating the processing of the human ear, the Auditory Image Model (AIM) has stood out as one of the most versatile. AIM provides both a functional model, which is faster, and a physiological model which more closely models the physical operation of the ear. Each of these models can be broken down into four primary processing stages (Figure 1). This chapter presented an overview of each of these stages for both models. A more in-depth look at the algorithms of the functional model, which were the focus of this research, are presented in Chapter 3 of this document.

### III. Theory

#### 3.1 Introduction

This chapter sequentially presents each stage of AIM in the order that data flowing through the model would encounter them. (Refer to Figure 1.) These stages are namely: outer/middle ear, filter bank, compression, adaptive thresholding and integration filtering. Each stage of AIM has two sections devoted to it in this chapter. The first section discusses the AIM implementation of the stage, while the second presents the work performed to simplify the processing needed to perform the required operations. In each case, tradeoffs were made which sacrificed some aspect of the model in a slight way. All of these differences are presented in the discussions that follow.

#### 3.2 Middle Ear Filtering

The first stage of processing accomplished by AIM models the outer and middle ear's transformation on the sound. Physically, the first stage includes the external ear, ear canal, ear drum, malleus, incus, and stapes (hammer, anvil, and stirrup) as shown in Figure 6. The function of the outer and middle ear is fundamentally that of a transducer that matches the impedance of the inner ear to that of the free air through which the sound arrives.

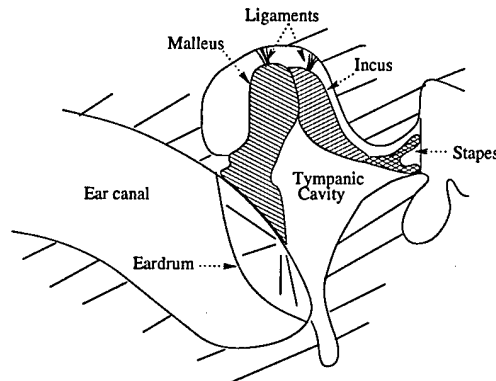


Figure 6: Anatomy of Middle Ear (after Lutman[2])

As the sound arrives, it travels through the middle ear canal where it causes the eardrum to vibrate. The motion of the eardrum is then transferred through the malleus, incus and stapes into the inner-ear fluids. Thus, the sound energy is transferred from a medium of air to a medium of liquid.

**3.2.1 Implementation in AIM.** The current model of the middle ear process originated in the 1962 work of J. Zwislocki[3]. Zwislocki modeled each organ of the middle ear as a combination of inductors, resistors, and capacitors. Through a process of experimentation and mathematical modeling, he derived an analog electric circuit model for the impedance matching of the middle ear. His circuit, (See Figure 7.) included 5 functional units, each with an electrical equivalent. The use of circuits allowed Zwislocki to reproduce the behavior of the differential equations of these organs, without numerically solving them.

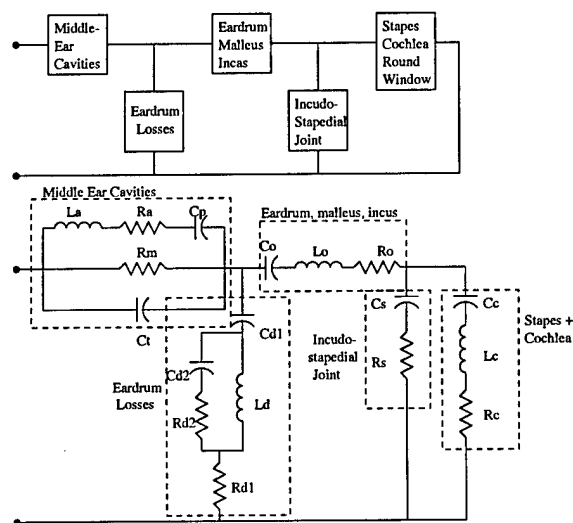


Figure 7: Zwislocki's Functional Model and Circuit Equivalent (after [3])

Zwislocki's work was revisited and expanded by M. Lutman and A. Martin in 1979[2]. In their research, Lutman and Martin adjusted some of the component values, but more significantly, they added components to model the stapedius muscle (not visible in the view of Figure 6). By including the stapedius muscle, they were able to simulate the effects of acoustic reflex. Through feedback from the brain, the acoustic reflex adjusts the middle ear transfer function to adapt to varying sound intensities.

Additionally, Lutman and Martin added a network of resistors and capacitors to the model which account for the transmission line effect of the ear canal. The addition of the ear canal to the model properly accounts for a sharp resonance region in the transfer function of the middle ear which reaches its peak near 3700 Hz. The peak, which can be seen in Figure 11, will be discussed in more detail in the next section.

C. Giguere and P.C. Woodland[11, 4], in their 1993 and 1994 work, took the model as proposed by Lutman and Martin, and transformed it into a digital network by means of the wave-digital filter (WDF). Additionally, they proposed replacing the inductor, which accounted for the cochlea in the model, with a transformer, as well as the addition of the external ear and the concha to the model. (See Figure 8.) Their model for the external ear does not account for the shape of the pinna, head, or upper torso; rather they begin their model at the ear opening.

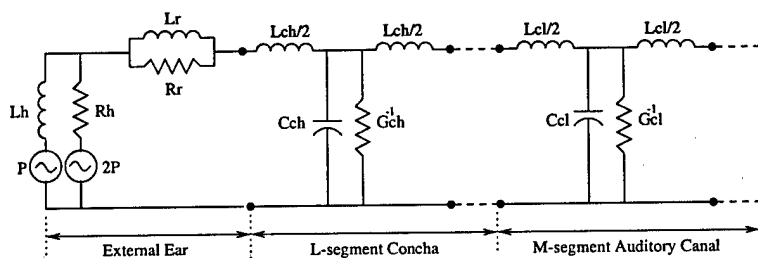


Figure 8: Giguere and Woodland's Outer/Middle Ear Model (after Giguere[4])

The replacement of the cochlea inductor with a transformer allowed them to expand the model into the cochlea where the cochlea also was represented with WDF transmission-line components. The resulting wave digital filters were then included into the AIM code as two separate modules: one for the outer/middle ear, and a second for the cochlea.

Giguere and Woodland's outer/middle ear WDF representation is the only model included in AIM for the first stage processing. It is interesting to note, however, that their wave digital filter representation of the cochlea is one of two models and is not used by default. Rather, AIM uses a filter-bank model of the cochlea by default. The filter bank approach to the cochlear model is the topic of a later section in this document.

**3.2.2 Approximation.** While the WDF approach to the outer/middle ear is physiologically very accurate, it is also very computationally expensive – requiring 39 multiplication and approximately 125 addition operations per data sample that enters the system.<sup>1</sup> At a sample rate of 20,000 samples per second, the WDF performs 780,000 multiplies and 2,500,000 addition/subtraction operations per second in order to be executed in real time.

<sup>1</sup>The code for the WDF is found in the AIM file `wdf_ear.c` found in the `wdf` subdirectory. The actual code is in the function `DoEarWdf()`.

Due to the complexity of the WDF, an approximation which would require fewer operations was desired. To replace the WDF, its impulse response was characterized and a digital filter with a similar impulse response was designed. The technique of impulse response approximation is sufficient because the impulse response of any system completely characterizes that system's transfer function.

The first step in the analysis of the Giguere-Woodland middle ear filter was to characterize the WDF. To characterize the WDF, an impulse was provided as input to AIM with all processing following the middle ear disabled via command line options. The output data from AIM was then captured and post-processed. The raw data of the impulse response is plotted in Figure 9. Post-processing was accomplished using Matlab<sup>2</sup> to convert the raw impulse response into its

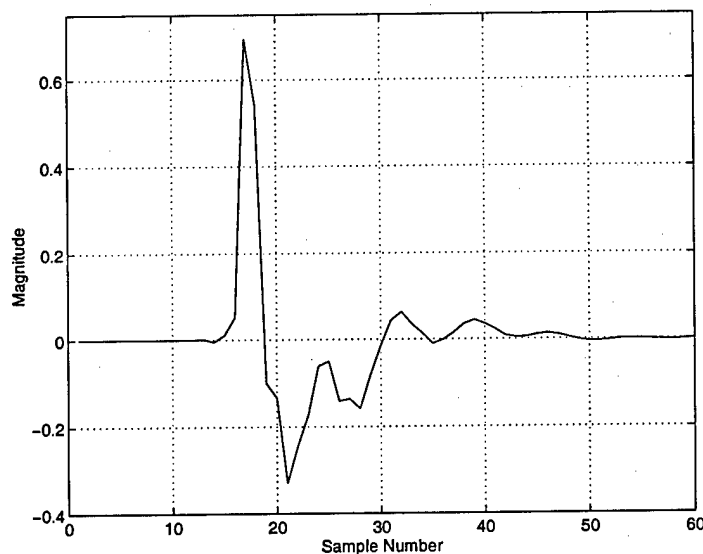


Figure 9: AIM's Outer/Middle Ear Impulse Response

corresponding frequency response as shown in Figure 10.

There is an obvious resonance near 2700 Hz shown in Figure 10. In order to positively identify which part of the middle ear model was causing the resonance, the electric circuit representation of the model was simulated. For the test, the complete outer/middle ear model given by Giguere (combined circuit using output from Figure 8 as input to Figure 7) was implemented

<sup>2</sup>Matlab is a registered trademark of Math Works Inc. Version 5.2 was used during for this research.

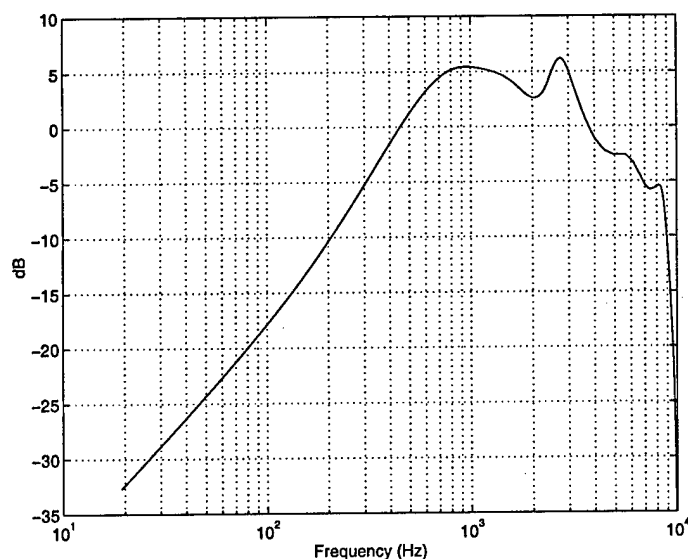


Figure 10: AIM's Outer/Middle Ear Frequency Response

using the Spice<sup>3</sup> simulation program. Figure 11 shows the results of these circuit simulation tests. Similarly, Figure 12 shows the Spice simulation results for the middle ear without the ear canal. A comparison of these two plots clearly shows that the resonance in the transfer function is due to the ear canal, not the middle ear structures. Interestingly, however, the exact location of the peak does not agree between the WDF of AIM and the circuit. The difference can be attributed to the fact that the Giguere WDF implementation includes the ear canal, ear opening, and the head modeled as a sphere. The original circuit did not include these outer elements.

If the resonance in Figure 10 is ignored<sup>4</sup>, it is easy to see that the resulting filter is a low-order band-pass filter centered near 1000 Hz. In fact, under closer observation, the rising edge of the filter is close to 40 dB/decade, while the falling end is closer to -20 dB/decade. The frequency response suggests a filter with 2 low frequency zeros and 3 poles near 1 kHz. Therefore a Matlab script was written which used a process of smart iteration to locate these poles and zeros so that the resulting frequency response was optimized using the method of least-squares fit. The resulting

<sup>3</sup>Spice (Simulation Program with Integrated Circuit Emphasis) is a circuit simulation program originally developed at UC Berkeley. There are now many versions sold by a variety of companies. Both HSpice by MetaSoft as well as Berkeley Spice version 3f4 were used.

<sup>4</sup>The resonance is an artifact of the ear canal, not the middle ear. More discussion on the resonance follows later in this section.

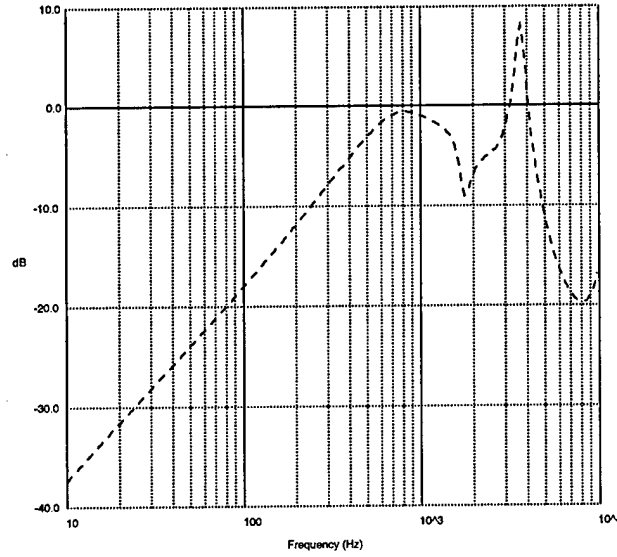


Figure 11: Spice Simulation of Outer/Middle Ear Including the Canal

S-domain filter equation was:

$$H(s) = \frac{26861(s + 270)^2}{(s + 2884)(s + 6640)^2}. \quad (3)$$

The resulting continuous S-domain expression of Equation 3 was then converted to a discrete Z-domain equation using the Matlab command *c2dm* with the zero-order-hold option selected and the sampling rate set at 20 kHz. The resulting difference equation for the Infinite Impulse Response (IIR) filter was:

$$Y_{(n)} = A_1 X_{(n-1)} + A_2 X_{(n-2)} + A_3 X_{(n-3)} + B_1 Y_{(n-1)} + B_2 Y_{(n-2)} + B_3 Y_{(n-3)} \quad (4)$$

where  $X_{(i)}$  are previous inputs,  $Y_{(i)}$  are previous outputs, and the constants were defined by

$$\begin{aligned} A1 &= 0.903479 & A2 &= -1.783521 & A3 &= 0.880206 \\ B1 &= -2.300689 & B2 &= 1.757066 & B3 &= -0.445659. \end{aligned}$$

Equation 4 was then coded in C as an AIM-compatible module and added to AIM as an option which could be selected using command-line options. To verify the accuracy of the IIR filter,



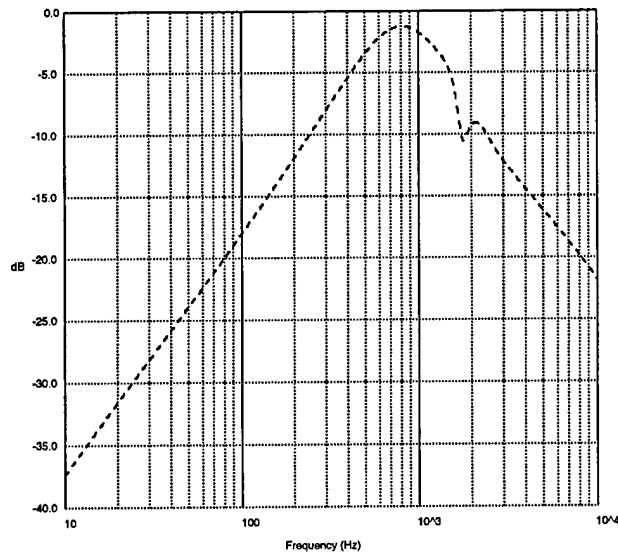


Figure 12: Spice Simulation of Outer/Middle Ear Without the Canal

the same impulse test used to characterize the original AIM outer/middle ear filter was duplicated using the approximate (IIR) model. Figure 13 illustrates a comparison between the AIM middle ear and IIR filter.

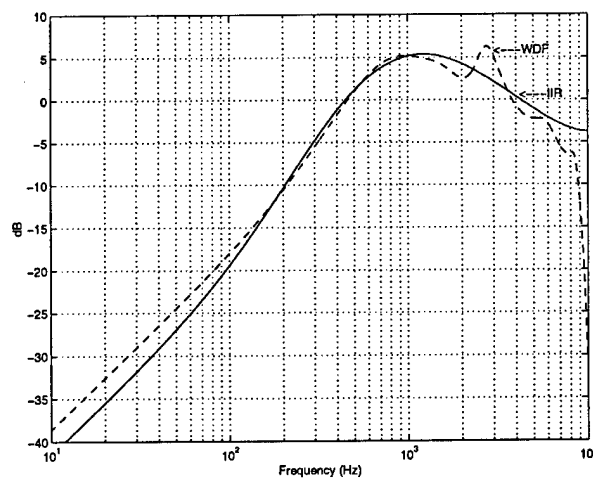


Figure 13: Comparison Between AIM and the Approximate Middle Ear Filter

Figure 13 clearly shows that the IIR equation has eliminated the resonance of the ear canal, which was intentional for the purpose of reducing the mathematical complexity. Although the removal of the effects of the ear canal may seem at first as unwarranted, for many applications the

sampled audio data is already filtered using a more accurate head-related transfer function (HRTF). These HRTFs include the effects of the shape of the head, upper torso, and pinna as well as the ear opening and ear canal. Additionally HRTFs can account for the angle at which sound arrives at the ear.

The inclusion of the effects of the ear opening and ear canal in experiments where an HRTF was already applied to the data set would subject the data to these effects twice. Therefore, in order to prevent subjecting the data to two different models of the ear canal, and to reduce the complexity of the computation, the resonance of the ear canal was removed from the model. The removal of the resonance was supported by phoneme recognition experiments discussed in Chapter 5.

Further testing compared the operational speed of the approximate IIR filter to that of the default WDF in AIM. In these tests, the sampled data for the spoken word "HAT" was replicated 10 times in a single file. The data file was then processed by AIM 6 times, 3 times using the WDF, and 3 times using the IIR filter. Testing was conducted on an unloaded Sun Microsystems SPARCstation-20 workstation running Sun OS 4.1.4 and timed using the Unix time command. Table 1 compares the user processing time for the IIR filter to that of the WDF with all other processing by AIM disabled by command-line options. As shown, the IIR filter algorithm is on

Table 1: Comparison of Filter Run Times (seconds)

Run #	WDF	IIR
1	26.28	6.87
2	26.02	6.96
3	26.31	6.93
AVERAGE	26.20	6.92

average 3.79 times faster than the WDF algorithm. The speedup is a reflection of the mathematical simplification of the approximation. It is appropriate to note that the data stream only passes through the IIR filter once.

Following the middle/outer ear filter (either WDF or IIR), the data is duplicated and passed through many channels in the filter-bank stage. Therefore, the speed-up noted here will not be observed when using more than one filter bank channel because the time required by the filter bank will dominate. The speedup of the middle ear filtering demonstrated here in software approxi-

mates the six-to-one ratio of multiplications between the two algorithms. When the IIR filter is constructed in application-specific hardware, the speedup will be even greater.

Having shown the spectral response of the IIR filter to be a close fit to the spectral response of the WDF (neglecting the WDFs resonance), the next step was to design a hardware implementation for the IIR filter. The goal was to reduce, if possible, the number of bits needed in the A and B constants and to determine the minimum number of bits of precision that would be needed internally for the filter to function correctly. The issue here is that the data stream entering the system will be 16-bit twos-complement integers. However, as the filters operate on the data, some amount of precision, or extra fractional bits, must be maintained in order for the digital filter to function.

To solve the storage size issue, the IIR filter was coded as a binary level simulation using the VHSIC Hardware Description Language (VHDL).<sup>5</sup> While VHDL allows simulation at any level of abstraction, in these tests the IIR filter was modeled at the gate level. Additionally, the word sizes were coded in generic terms thus allowing the model to be easily re-configured during the testing. As a result, direct comparisons could be made noting how the system functioned using different word sizes.

The VHDL testing revealed that 8 bits for the multiplier constants (3 bits for the signed integer portion, and 5 for the fractional part) were sufficient for the filter to function based upon its frequency response curve. Additional testing showed that 6 bits to store the fractional part of the intermediate results were near optimal. Increasing the number of bits beyond 6 had virtually no effect on improving the response of the filter, but fewer than 6 degraded the filter's performance. The response of the filter was determined using Matlab to post-process the filters response to an impulse input.

Although an 8-bit word was shown to be near optimal for the constants, the word size recommended later is 16 bits for the multiplier constants (3 bits of two's complement integer and 13 bits of fraction). The added precision was a byproduct of optimizations in the filterbank stage. Because the IIR filter and the filterbank were combined in the hardware implementation, the IIR filter benefited from the higher requirement of the filterbank. In the same manner, the filterbank requires

---

<sup>5</sup>VHDL is an industry standard language (IEEE Standard 1079-1987, revised in 1993) for the modeling of digital systems. VHDL is currently required for all new DOD contracts which involve digital systems.

8 bits of fraction for internal precision. Therefore, again the IIR filter precision was extended to allow the re-use of the hardware components. The filterbank mentioned here is discussed in the following section.

### 3.3 *Spectral Analysis*

The processing stage following the outer/middle ear filtering, both physiologically and in the model, takes place in the organ of the cochlea. In this organ the sound energy travels through a spiral shaped fluid-filled chamber where it stimulates hair cells which in-turn stimulate the nerves. These cells produce the electrical signals that are transmitted to the brain for processing via the auditory nerve. As the sound energy travels through the cochlea, a natural separation of the energy into the frequency domain occurs. The higher frequency energy is absorbed early in the cochlea, while the lower frequency energy propagates deeper into the spiral-shaped organ.

Because of the physical separation of the frequencies in the cochlea, the hair cells located closer to the oval window (where the energy enters the cochlea) respond to high frequency energy, while those cells located further from the oval window respond only to lower frequency energy. Therefore, the messages sent to the brain are in fact frequency-coded showing the spectral content of the sound energy. The separation of the sound energy into frequencies by the cochlea is nearly a continuous function due to the many thousands of hair cells.

**3.3.1 *Implementation in AIM.*** Because we must limit the size and complexity of simulation models, any feasible representation of the cochlea must use a finite number of discrete frequency bands to represent what we perceive as the otherwise continuous response of the cochlea. Much research has been applied to choosing the optimal shape for such a discrete filter[18, 16].

At first glance, one solution to the frequency separation problem appears to be the Discrete Fourier Transform (DFT). The DFT solution, however, is not acceptable because of the loss of all phase information; the DFT only provides energy content information. The hearing process depends heavily upon the phase angle of the incoming sound for localization of the sound source. Removing phase information prohibits a model from being able to do localization processing[25]. Therefore the DFT is unacceptable as a solution to accurately model the hearing process.

The current model is based on the *revcor* function. The *revcor* (REVerse CORrelation) function is a continuous representation of a set of data points obtained from an experiment in which the firings of a primary auditory fiber are correlated with the waveform entering the ear[16]. The result of the reverse correlation process is an approximation to the impulse response of the ear.

At the time, researchers looking for an analytic expression to model the *revcor* function identified the gammatone function as a good approximation[14]. The gammatone gets its name from the fact that the impulse response is a cosine wave (tone) with the amplitude envelope shape of the gamma function found in statistics. The impulse response of the gammatone function is shown in Figure 14. Expressly, Patterson[16] gives the gammatone equation for ( $t \geq 0$ ) as:

$$gt(t) \propto t^{(n-1)} e^{(-2\pi bt)} \cos(2\pi f_o t + \theta) \quad (5)$$

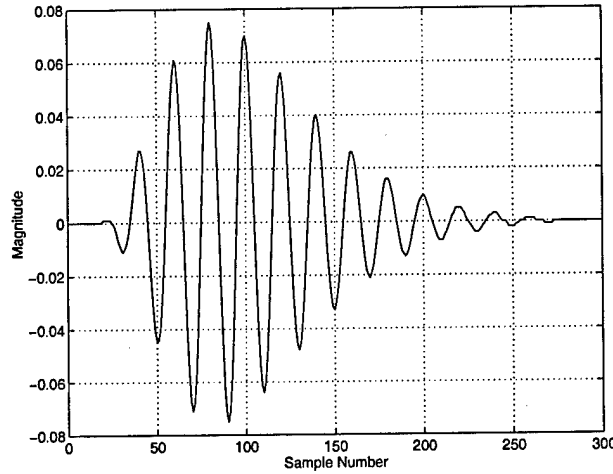


Figure 14: Impulse Response of 4<sup>th</sup> Order 1 kHz Gammatone Filter

In Equation 5,  $f_o$  represents the center band of the filter,  $n$  the order of the filter (most commonly fourth-order for auditory modeling),  $b$  is a parameter descriptive of the bandwidth of the filter, and  $\theta$  an initial phase angle. A thorough discussion on the choice of the gammatone filter as well as the filter spacing and bandwidth can be found in [18, 20, 21].

Equation 5 may be transformed into the frequency domain by noting that the multiplication of the gamma function by the cosine will result in the convolution of the frequency spectrum of the

gamma function with the impulse frequency response of the cosine. Thus the frequency domain expression becomes (ignoring the phase term because it does not affect the shape of the function and considering only positive frequencies):

$$GT(f) \approx [1 + j(f - f_o)/b]^{-n} \quad (6)$$

From Equation 6 it is easy to see that the filter can be implemented by cascading  $n$  identical filter stages. (As in Equation 5,  $n$  represents the filter order.) To optimize the filter for implementation in a C program (namely AIM), the input data is first down-shifted by  $f_o$  through multiplication with a complex exponential. The complex multiplication relocates the filter's center frequency to DC. The data is then processed using  $n$  passes through a first order gammatone low-pass filter. Finally, the data stream is up-shifted back to  $f_o$  again by multiplying with a complex exponential. One should recognize this process as the classic low-pass prototype filter method.

A computational cost analysis was done in considering the frequency shift approach just discussed as an alternative to the filtering requirements. The initial down-shift of frequency requires a complex number multiplication, the cost of which is actually two multiplies. To further increase speed, the values for the complex representation of the shift frequency need to be stored in read-only memory (ROM), or require several additional multiplies to produce them as needed.

One possible solution to the storage versus multiplication trade-off considered was to change the filter spacing. The frequency shift method of AIM is based upon the multiplication by complex coefficients which lie equally spaced on a unit circle. The location of the complex values on the unit circle is determined by the ratio of the sampling frequency to the center frequency of the filter. Therefore, by spacing the filters at octave multiples above a base set, the coefficients from the first set may be used for the higher frequency filters.

The reuse of one set of ROM coefficients was tested and proven effective through modifications to AIM, however, it was not pursued beyond proof of concept since the resulting filter spacing is not consistent with the ERB developed by previous researchers.

Continuing with the cost analysis, the actual filter algorithm must be performed on a complex value as a result of the frequency down-shift. Again, if we assume a ROM storage for the needed coefficient, the filter algorithm requires one complex multiplication and two complex additions.

Therefore the actual number of operations for a fourth-order gammatone filter will be 8 ordinary multiplications and 16 additions per data point.

Finally, the filtered data must be up-shifted back to  $f_o$  at the cost of another complex multiplication costing two real multiplications and one addition. Thus, the total cost of the recursive gamma-tone filter as implemented in AIM (assuming all constants are stored in ROM) is 12 multiplications and 17 additions. The amount of ROM storage needed is related to the sampling frequency ( $f_s$ ) and the center frequency ( $f_o$ ) of the filter. Approximately  $f_s/f_o$  locations are required for each filter in the filter bank.

**3.3.2 Approximation.** Because of the number of filter elements required in the filter bank to obtain satisfactory frequency resolution for speech processing research, the filter bank quickly becomes the bottle neck in the auditory model. Therefore, alternatives were sought to reduce the amount of work required. The solution chosen was based on the all-pole gammatone filter (APGF) approximation described by Malcolm Slaney[17]. In his paper, Slaney was the first to show that the Laplace transform of the fourth-order gammatone function produces an  $S$  domain equation with four real zeros and four complex conjugate pair poles. He further shows that the only significant effect the four zeros have on the filter is to control the filter attenuation near DC. Therefore, he proposes the elimination of the zeros resulting in an  $S$  domain equation for a first order all-pole gammatone filter of:

$$GT(s) \approx \frac{1}{(s + B)^2 + \omega^2} \quad (7)$$

where  $B$  is a bandwidth term and  $\omega$  is the center frequency in radians/second.

Equation 7 can be easily transformed into the discrete domain and put into the form of a filter difference equation. The transformed first order APGF equation thus becomes:

$$Y[n] = aX[n-1] + bY[n-1] + cY[n-2] \quad (8)$$

where  $Y[n]$  is the current filter output,  $Y[n-1]$  and  $Y[n-2]$  are the previous two outputs, and  $X[n-1]$  is the previous input. The constants  $a$ ,  $b$ , and  $c$  are easily computed based on the center frequency, bandwidth, and sampling rate for the filter. (See Appendix A for computation of these constants.)

As was the case for the original gammatone filter, a fourth-order all-pole filter can be obtained by cascading four identical filters together.

From Equation 8 it is easy to see that only three ROM constants per filter are required and that a fourth-order filter requires 12 multiplies and 8 additions. While the number of multiplies is the same as that of the recursive gammatone of AIM, the number of ROM values required for the APGF is significantly smaller than that of AIM. In AIM a look-up table is used to hold the complex phaser for the frequency down-shift. The size of the AIM phaser table is  $5/4$  times the sampling frequency divided by the filter's center frequency and is used to hold both the sine and cosine values. One of these tables is required for each filter. In addition to a large savings in ROM storage, the number of additions has been reduced from 17 to eight.

To test the viability of the APGF, the filter was coded into the C programming language as an AIM module and added to AIM as an option for the filter bank processing. Testing was then completed to compare the impulse response of AIM using the original filter to that of the APGF. Results of these tests are shown in Figures 15 and 16.

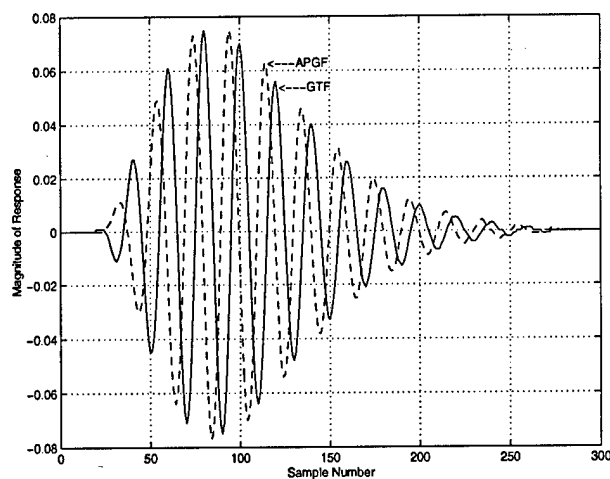


Figure 15: Comparison of Gammatone and APGF Impulse Response

The response of Figure 15 was generated by processing an impulse of magnitude 1000 through AIM (using the "genbmm" command) with only one active filter in the filter bank. While many different frequencies were tested, in the case shown the filter center frequency was 1 kHz because it falls near the center of the filter bank. To ensure the correctness of the experiment, the



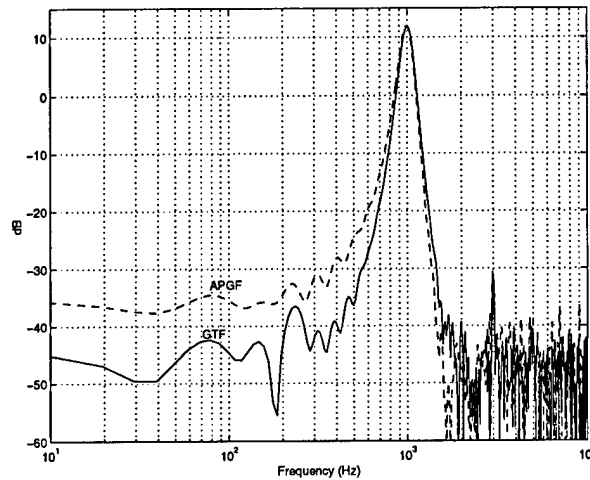


Figure 16: Frequency Response of Gammatone and APGF

middle ear filter was turned off. The output data was written to a file, then processed in Matlab where both the time and frequency domain representations were compared.

In Figure 16, the data used to generate Figure 15 was further processed and plotted using Matlab to convert the raw data into the frequency spectrum. Figures 15 and 16 demonstrate that the APGF differs slightly from the gammatone filter in both the time and frequency domain. In the time domain (where AIM does its processing), the difference is a phase shift. Within any given filter from the filterbank, the phase difference manifests itself as a time shift, and is constant. For the filter shown in Figure 15 the delay is under 0.5 ms. Because current speech recognition research averages the data in 16 ms windows[25], it is reasonable to discount these small time shifts.

The differences in the frequency domain, as seen in Figure 16, agree with those predicted by Slaney, namely the loss of attenuation near DC. As claimed previously, the difference can be discounted because of the middle ear filter (which was disabled in the generation of Figure 16). When the effect of the middle ear filter is included prior to the spectral filtering, the difference between the low-frequency responses of the APGF and the gammatone filter is decreased. Figure 17 illustrates the response of the APGF and the gammatone filters with the corresponding middle ear filters enabled.

A final comparison between the AIM gammatone filter bank, and the APGF filter bank addressed the speed of the code. Although the actual execution time of the two different algorithms in AIM is not expected to map directly to the execution time of an optimized hardware implemen-

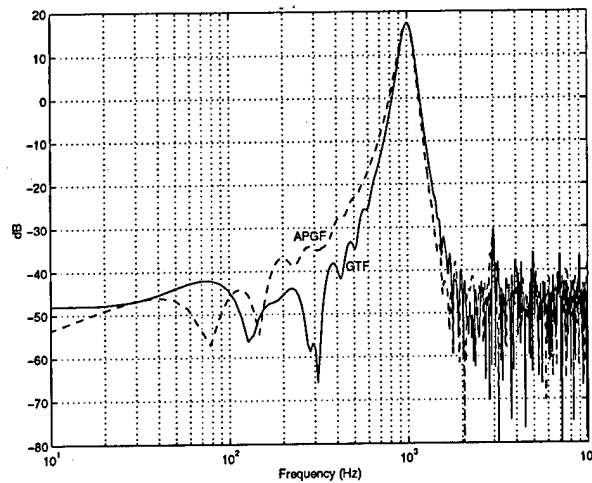


Figure 17: Gammatone and APGF Frequency Response with Middle Ear Filtering

tation, such a comparison is valuable as an indication of the number of operations that must be computed. Table 2 compares the run times of AIM using the default gammatone filter (GTF) and the new all-pole gammatone filter (APGF).

Table 2: Comparison of Filter Bank Run Times (seconds)

Run #	GTF	APGF	APGF4
1	9.58	9.25	6.10
2	9.58	9.22	6.13
3	9.53	9.27	6.13
4	9.56	9.21	6.12
AVERAGE	9.56	9.24	6.12

As can be seen in Table 2, a variant of the APGF (named APGF4) was developed and tested. The APGF4 filter was a fixed fourth-order filter mathematically equivalent to the APGF. The difference between the APGF and the APGF4 is in their implementation. The APGF is a recursive filter. In APGF4, four recursions of the APGF were algebraically combined into a single fourth-order expression. The single expression was then implemented without the need for recursion or looping. While the APGF4 executes faster on a general purpose processor, it requires much higher precision than is possible using fixed-point integer arithmetic. Therefore, the APGF4 was not further pursued as an alternative in the hardware solution.

For the tests run in Table 2, a seven-second audio stream was processed through AIM with the outer/middle ear filter turned off as well as all processing following the filter bank. Disabling the mentioned functional blocks isolated the filter bank from the rest of AIM. In each case there were 64 filters in the filter bank. The processing was accomplished on an unloaded 266 MHz Intel Pentium-Pro processor running Linux 2.0.34. The AIM code was compiled for Linux using the GNU *gcc* compiler tools. Times recorded are the user CPU times reported by the operating system's *time* command.

As expected, because of the reduction in arithmetic operations, the APGF bank operates with a slight advantage in speed over the gammatone filter bank. The increase in speed was found to be dependent upon the processor and operating system. When run under Sun-OS 4.3.14, an increase in speed of 1.3 times was realized for the APGF over the GTF. Further improvements in speed are expected when the APGF is implemented in an application-specific architecture.

Having selected the filter equation, the next step was to optimize the filter for hardware implementation. Assuming a fourth-order filter, an initial analysis would indicate the need for 3 read-only memory (ROM) constants and 12 read-write memory (RWM) cells per frequency channel. However, upon closer inspection, a data dependency appears within each channel where the output of any given stage of the filter bank becomes the input to the following stage. Taking advantage of spatial locality of the data in the algorithm, the number of RWM cells can be reduced from three cells to two per recursion of the filter thus reducing the total storage requirement per filter to 3 ROM and 8 RWM cells.

The RWM reduction is possible because the input value for second through fourth recursions of any filter in the bank is the output of the recursion that preceded it. Since the previous recursion will have just completed, there is never a need to store its output to memory; the output can be re-circulated directly into the following filter. The input to the first stage of any filter in the bank will always be the output of the IIR middle ear filter, which is stored separately.

Next to be considered was the precision, or word sizes, for these memory locations. Again, VHDL was employed to model the filter at the binary level so that realistic comparisons could be made between systems operating with different word sizes.

For the VHDL tests, a behavioral description of the APGF was written. Although the model was behavioral, i.e. the algorithm appeared in much the same form as in the C program code, the data objects were generic-sized vectors of bits. Abstract data typing and overloading of the math operators allowed these vectors to be manipulated using standard algebraic notation while preserving the binary behavior of an integer-based system.

The actual testing proceeded much as the testing for the middle ear word sizing experiments. The bit size of the stored data was varied, as well as the number of additional bits which were stored as fixed-point fraction bits. For each case, an impulse was run through the filter and the output was collected and analyzed using Matlab to post-process the data and generate frequency response plots. The output data was also viewed directly in the time domain and compared to the time domain impulse response of the AIM filters.

When considering the ROM word size, the first step was to generate a table of possible constant values that would be needed for a typical system. A table of constants was generated using the *info=all* option of AIM with the APGF selected. The table revealed that the absolute value of the largest constant was less than 2.0. Therefore, the constant needed only 2 bits to represent the integer: a sign bit and a single data bit, leaving the number of fractional bits in the ROM value as a single degree of freedom for the experiment.

Because the final system was constrained to accept 16-bit integers and provide 16-bit integer results (the constrain imposed by the requirement to remain compatible with AIM), it was natural to internally store the integer portion of the values in 16 bits. Therefore, as with the ROM constants, for the RWM storage there was again a single degree of freedom left for consideration: the length of the fractional part.

A series of experiments were run where the ROM fraction size was varied from 6 to 16 bits and the RWM fraction size was varied from 0 to 12 bits. As previously mentioned, at each step an impulse was applied to the digital filter and the spectral shape of output data was compared to that of the AIM filters using Matlab. For these experiments, a filter element with a center frequency of 100 Hz was used because: 1) it is the minimum frequency used by default in AIM, and 2) its constants are the smallest, making it the first filter element to deteriorate due to round-off errors.

Table 3: Effect of Word Size on Filter Frequency

Bits of Precision	Percent Shift
10	15
12	15
14	8
16	8

The testing revealed that if fewer than 10 bits are used, the first ROM constant for the 100 Hz filter is rounded to zero and the filter does not function. With 10 bits, the same constant realizes a single '1' in its least significant bit position. While the filter will function with 10 bits, its operation is marginal. In fact, the center frequency of the filter is shifted by 15% toward DC. Increasing the number of ROM fraction bits reduces the undesired frequency shift. Table 3 illustrates the relationship observed between the number of bits and the shift of the filter's center frequency for the 100 Hz filter.

When the ROM word is set to 16 bits, 2 for the integer part and 14 for the fixed-point fraction, the worst case frequency shift is reduced to 8%. That is, the 100 Hz filter behaves as though it were centered at 92 Hz. Extending the ROM word size an additional 2 bits (16 bits of fraction) did not significantly change the frequency shift of the response from that of 14 bits. Because 16 bits is a standard word size and the next logical increase in size did not significantly improve the filter response, a 16-bit word size (2 integer and 14 fraction) was chosen for the ROM coefficients. It is significant to note here that regardless of word size, the frequency shift error is inversely related to the filter center frequency. The higher the filter's designed center frequency, the smaller the frequency shift error.

Even with as much as 8% error in the location of the lower frequency filters, it is still possible to precisely specify a filter. For example, to locate a filter at 100 Hz, an impulse can be processed through AIM with a 100 Hz filter specified. The output is then processed (using Matlab) to determine the resulting filter center frequency. The difference between the actual frequency and the desired frequency is then added to the desired frequency and used to specify a second run of AIM. Through iteration coefficients can be chosen which give the desired frequency response.

For the RWM word size, the key figure of merit was the time domain response of the filter. If too few fraction bits were maintained, the time domain response possessed a negative DC bias.

As would be expected, the bias effect also showed up in the frequency domain as a positive DC gain.

The DC shift of the frequency response was directly influenced by the number of fractional bits in the RWM word. Testing revealed 8 bits of fraction to be the optimal size for this application. Therefore the RWM word size of 24 bits (16 for the two's-complement integer and 8 bits for fraction) was selected.

At this point, all that remained for the APGF was to design an architecture that would take advantage of the spatial locality of the data and use the optimized word sizes. The details concerning this architecture are discussed in Chapter 4.

### 3.4 *Neural Encoding—Rectification and Compression*

In the Auditory Image Model, after the sound has been filtered by the gammatone filter bank, which models the motion of the basilar membrane, it enters the neural encoding stage. The first step of neural encoding is half-wave rectification and logarithmic compression of the data. The half-wave rectification models the way in which the hair cells respond to a compressive stimulus. The compression models the way in which the large movements of the basilar membrane are transformed into the smaller nerve firings[26].

Following the rectification and compression of the data, AIM models the firing of the hair-cell nerves through a process coined adaptive thresholding. While the rectification and compression, and the adaptive thresholding are actually both part of the neural encoding process, these topics have been split into two sections for clarity. The remainder of Section 3.4 presents the discussion on rectification and compression, while Section 3.5 presents the adaptive thresholding.

**3.4.1 *Implementation in AIM.*** In AIM, the half-wave rectification is accomplished by simply setting any negative value in the data stream to zero before the compression. The compression is then performed by converting the output of the filter stage to millibels (one millibel = 100 decibels). The algorithm used for the compression to millibels first computes the logarithm (base 10) on the data, then multiplies the result by 2000. The logarithm employed by AIM is the  $\log_{10}(X)$  function from the standard C libraries.

**3.4.2 Approximation.** As noted in Chapter 2 , researchers have proposed and utilized a variety of compression algorithms. Currently, the logarithmic method employed by AIM as its default is the most widely used; thus it was chosen for implementation. The focus of this research was to develop a method for efficiently converting the data stream to millibels[27].

In 1962, John Mitchell reported a linear approximation technique for quickly computing the logarithm in base two[28]. In his technique, to compute the approximate  $\log_2$  of a number, Mitchell linearly interpolated between the  $\log_2$  of the two nearest powers of two. For example, to compute  $\log_2(6)$ , one would linearly interpolate between  $\log_2(4)$  and  $\log_2(8)$ . Figure 18 illustrates Mitchell's technique graphically over one log cycle from 32 to 64.

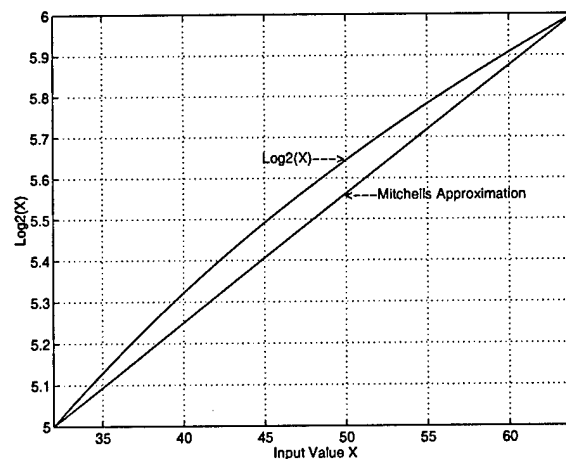


Figure 18: Mitchell's Linear Approximation to  $\log_2$

Perhaps Mitchell's greatest contribution was his technique for performing the linear interpolation. His technique began with finding the characteristic, or whole number part, of the logarithm. To determine the characteristic, Mitchell counted bit positions to identify the bit number corresponding to the most significant non-zero value in the input binary number. For example, consider the 8-bit computation of  $\log_2(55)$ . The binary representation of the integer 55 is  $00110111_2$ . The most significant one is located in bit position 5 (bit 0 being the right-most bit), therefore the characteristic of  $\log_2(55)$  is 5.

After the most significant one had been located, Mitchell extracted all of the bits of lower significance and made them into a fixed point fraction appended behind the characteristic. Contin-

uing with the example of  $\log_2(55)$ , the approximate logarithm (base 2) is  $101.10111_2$  or 5.718750. The actual  $\log_2(55) = 5.78136$  (rounded) indicating an error, in this case, of -1.08%.

Mitchell's algorithm always has a maximum magnitude error of -0.08496 on the values that fall exactly between two powers of two[28]. Therefore, the worst case percent error occurs when the input value is 3 (exactly between  $2^1$  and  $2^2$ ). In this case the error is approximately 5.36%. The percent error is cyclic and decays exponentially as shown in Figure 19.

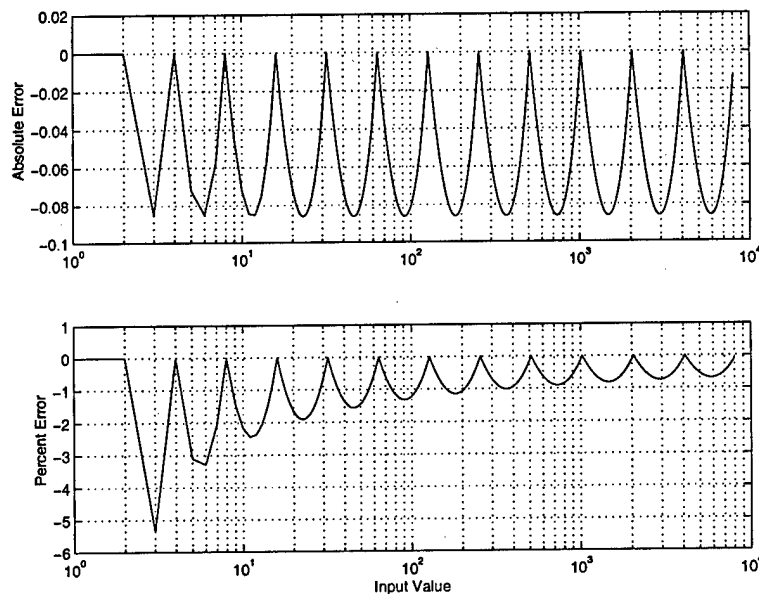


Figure 19: Percent Error in Mitchell's Approximation

While others have reported improvements to the accuracy of Mitchell's algorithm, they have done so at the cost of added complexity[29, 30]. Specifically, the algorithms of [29] and [30] require the calculation of fractions of the input value which are not powers of two. For this research, a method was desired that had better performance in the small input range than Mitchell's algorithm, with less complexity than what others have reported.

A technique to reduce the error came through careful study of the approximation and the percent errors as shown in Figures 18 and 19. Note particularly that the magnitude of the error repeats cyclically between powers of two, and that the maximum error is always -0.08496. A method to reduce the error is to break each log cycle into two sub-regions and change the slope of Mitchell's approximation so that the mid-point of the approximation is pulled closer to the actual log value.



The proposed technique is in fact similar to the techniques of Combet[29] and Hall[30]. In their approaches, they generalize to more than two sub-regions and develop precise mathematical equations which minimize the error. The disadvantage of the algorithms developed by Combet and Hall is the need for the computation of fractional multiples of the input value, as well as several conditional additions and subtractions.

The method developed in this research focuses on the binary representation of the numbers. Considering Mitchell's approach as described above, he found that the input number itself contained a linear approximation between two known points. The question posed was whether a similar simple approach to adjust the approximate log existed. The answer is yes, and as in Mitchell's algorithm, the solution is contained within the input value.

To understand the new approximation, recall the magnitude of the error in any cycle as seen in Figure 18. The absolute value of the error is 0.08496 which has a binary representation of  $0.000101011011_2$ . In order to zero-out the midpoint error, we need to add this error value to our approximate log. However, adding 0.8496 only applies when the input value is exactly between any two powers of two. For other inputs some fraction of this value would need to be added.

Now, consider the binary representation of the input values near the mid-point of a log cycle. For example, 12 which appears between 8 and 16 and has the binary form (in 8 bits) 00001100<sub>2</sub>. Using Mitchell's approximation algorithm above,  $\log_2(12) \approx 11.100_2$ . We know also that the exact value of  $\log_2(12) = 11.100101011011_2$  (the sum of our approximate value and our maximum error). Taking into account the weight of each bit in the fraction, it is easy to show that the single "1" bit in the 4<sup>th</sup> position to the right of the decimal point ( $0.0001_2$  which represents 0.0625) actually accounts for 73.6% of the error in Mitchell's approximation. Therefore the fourth fractional bit is the target of the new method.

When considering how the fractional part of Mitchell's approximation was generated, we see that it is a linearly increasing value. Thus, by copying the fractional part of Mitchell's approximation, shifting it to the right by 3 bit positions, and adding it back to the approximation, an adjustment factor is computed that linearly increases and at the mid-point will have corrected 73.6% of the original error. Effectively, the copy-shift-add correction changes the slope of the ap-

proximation so that it approaches the actual log at the mid-point between powers of two. Table 4 tabulates the correction for the input values 8 to 12.

Table 4: First Correction to Mitchell's Approximation

Input	Binary Form	Mitchell	% Error	Adjusted	% Error
8	00001000	011.00000000	0.00	11.00000000	0.00
9	00001001	011.00100000	-1.42	011.00100100	-0.92
10	00001010	011.01000000	-2.16	011.01001000	-1.22
11	00001011	011.01100000	-2.44	011.01101100	-1.09
12	00001100	011.10000000	-2.37	011.10010000	-0.63

The shift-add does not entirely solve the problem, however, because now the adjusted approximation and the actual log are converging at the cycle midpoint. Therefore the slope of the approximation must be adjusted downward after the midpoint in order to keep the approximation inside the log curve. Without the second correction, the approximation curve would penetrate the actual log curve resulting in a positive, and even greater error between the midpoint and the upper endpoint.

Again, a simple adjustment was found within the approximation. To identify the solution, note first that the change in slope must occur when the most significant bit of the fraction is set to a 1 indicating that the input is greater than half of the distance between two consecutive powers of two. Second, the complement of an increasing sequence of binary numbers is a decreasing sequence. For example, observe the two-bit count: 00 01 10 11, which represents the sequence 0 1 2 3. The complement of the binary sequence is 11 10 01 00, or 3 2 1 0 which is clearly a down count.

Now observe the first four bits of fraction from Mitchell's linear approximation near the midpoint of any log cycle. The sequence (disregarding for the moment the lower-order bits) would be:

$$.0110_2 \quad .0111_2 \quad .1000_2 \quad .1001_2 \quad .1010_2$$

where the  $.1000_2$  is exactly between two powers of two. For the first correction just discussed, these bits were simply copied, shifted to the right, then added back to the sequence. However, if the complement of the bits is computed when the left most bit is set, then the correction factors

become:

$$0110_2 \ 0111_2 \ 0111_2 \ 0110_2 \ 0101_2 \quad (9)$$

From the sequence of Equation 9 it is easy to see that the correction factor will begin to decrease linearly until the fraction is one step from the end of the log cycle. At that point the fraction bits would be  $.1111_2$ , and consequently the correction would become  $0000_2$ . Hence the approximation converges linearly to an exact value at the end of the log cycle.

Table 5 tabulates the remaining values in the log cycle that was started in Table 4 using the second correction. Note that the value for 12 has been changed between the two tables because it is the first value where the most significant bit of the fraction is set.

Table 5: Second Correction to Mitchell's Approximation

Input	Binary Form	Mitchell	% Error	Adjusted	% Error
12	00001100	011.10000000	-2.37	011.10001111	-0.74
13	00001101	011.10100000	-2.04	011.10101011	-0.88
14	00001110	011.11000000	-1.51	011.11000111	-0.79
15	00001111	011.11100000	-0.82	011.11100011	-0.52
16	00001000	100.10000000	0.00	100.00000000	0.00

Several observations can be made from Tables 4 and 5. First, the percent error is always negative, suggesting a positive offset could be added to further improve the approximation. Second, as can be seen in Table 5, the number of bits included in the adjustment will affect the final result. Finally, Table 5 changes the approximation of the mid-point value, and in fact increases its error. In order to better understand all of these effects, the algorithm was modeled using a VHDL simulation.

In the simulation, the number of bits to be included in the shift/add, as well as the number of bit positions these bits were to be shifted was left as generic (changeable) parameters. Simulations were then run for integer inputs ranging from 1 to 8000, while the number of bits and shift positions were varied. Two parameters were measured during the testing: the RMS error over the entire range, and the worst case percent error. Table 6 summarizes the results of the VHDL testing.

As can be seen in Table 6, the minimum percent error and the minimum RMS error do not occur simultaneously. Fortunately, both of these minimums occurred when the number of

Table 6: VHDL Results for Approximation Adjustment Strategies

# Bits	Positions of Right Shift					
	2		3		4	
	RMS	MAX %	RMS	MAX %	RMS	MAX %
6	0.1389	+2.25	0.2665	-1.75	0.4188	-3.25
5	0.1361	+2.0	0.2751	-1.75	0.4231	-3.5
4	0.1385	+1.5	0.2928	-2.0	0.4318	-3.5
3	0.1717	-1.0	0.3301	-2.5	0.4501	-4.0
2	0.2913	-2.0	0.4094	-3.5	0.4895	-4.25

shift positions was two.<sup>6</sup> Therefore, a two position shift is the optimal selection for the shifting distance. For the number of bits to include, the best choice is four bits because it falls between the optimal choice for the minimum RMS error (3 bits) and the minimum maximum error (5 bits). The resulting approximation exceeds the actual log for some values which works to both reduce the worst case error as well as the overall RMS error. Figure 20 compares the resulting approximation to the actual logarithm base 2 over the input range of 32 to 64.

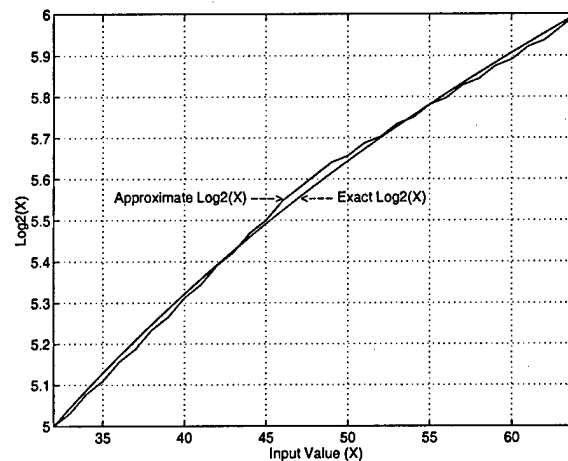


Figure 20: Comparison of Adjusted  $\log_2$  Approximation and Actual  $\log_2$

Because the logarithm approximation is cyclic, the curve shown in Figure 20 is replicated between every pair of consecutive powers of two. It is interesting to note that the approximation is no longer linear as suggested earlier in the development of the theory. The non-linearities in the approximation are an artifact of the addition of the adjustment value to the original approximation.

<sup>6</sup>Although not shown in the table, the test included a shift of only one bit position. Because of the magnitude of the error with only a single shift, these results were not included.

In the case considered as an example in Tables 4 and 5, there were no logical one bits in the original fraction where the adjustment factor was added. However, as the input value increases, logical one bits begin to appear in these positions of lower significance. As a result, an interference occurs which causes the slope to increase or decrease faster than desired.

The bit interference occurs in much the same way as two waves of water that collide on an angle. That is, there are times where they add constructively resulting in a larger signal, times where they add destructively resulting in a smaller signal, and times when they cancel completely. The net result is the oscillating effect that is observed in Figure 20.

Figure 21 illustrates the magnitude error and percent error of the adjusted and optimized approximation algorithm. In these figures, the oscillations just mentioned appear more pronounced because the magnitude of the actual logarithm has been subtracted. These oscillations have very little effect on the performance of the auditory model because: 1) their magnitude is small relative to the input signal strength, and 2) it would take a very low frequency (below the frequency cutoff of the middle/outer ear filters) in order to produce samples closely enough spaced in magnitude to see the effect.

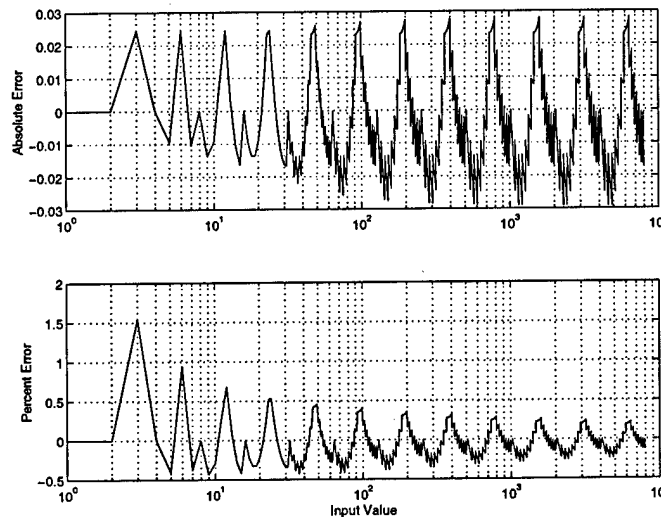


Figure 21: Magnitude and Percent Errors of Optimized Approximation

An attempt was made to further improve the accuracy of the approximation. In the trial, the adjustment process just discussed for the 4<sup>th</sup> bit of the binary fraction was repeated for the 6<sup>th</sup> bit of the fraction. That is, the most significant bits of the fraction in Mitchell's approximation were

again copied, shifted, then added into the approximation. The shifting of the bits in the second adjustment was further to the right than in the first case. Optimizing was accomplished using VHDL in much the same way as was described in the preceding paragraphs.

The second adjustment reduced the peak error of the approximation to 1.09%. However, because the neural encoding which follows requires a minimum of 1024 millibells for a nerve to fire, the peak error seen by the neural algorithm is always under 0.5% independent of whether or not the second approximation is applied to the logarithm. Since the smaller inputs (which produce the largest logarithm error) are rejected in the neural stage, and because the second adjustment increased the hardware complexity, the second adjustment was not implemented.

While perhaps necessary, the computation of the logarithm in base two does not yield the result required, that is millibells. Fortunately, the remaining conversion can be accomplished through carefully selected addition operations. To illustrate the conversion to millibells, consider the mathematical identity.

$$\log_2(X) = \frac{\log_{10}(X)}{\log_{10}(2)} \quad (10)$$

From Equation 10 it is easy to see that:

$$\log_{10}(X) = \log_2(X) \times \log_{10}(2). \quad (11)$$

Now recall the definition of decibels (dB):

$$dB = 20 \times \log_{10}(X) = 20 \times \log_2(X) \times \log_{10}(2). \quad (12)$$

To convert dB into millibells, we simply multiply by 1000 yielding the expression:

$$millibells = 2000 \times \log_{10}(2) \times \log_2(X) \approx 602.06 \times \log_2(X). \quad (13)$$

A quick approximation for the  $\log_2(X)$  has just been derived leaving only the multiplication by 602.06 (which is rounded to 602) to complete the conversion of the input value X into its equivalent millibel form. Fortunately, because the multiplier value is a constant, the multiplication can be reduced to a few addition operations. The implementation of Equation 13 is discussed in detail in Chapter 4.

### 3.5 Neural Encoding-Adaptive Thresholding

The second part of the neural encoding process is the simulation of the activity of the nerve cells in the cochlea. The nerves in the cochlea, commonly referred to as the hair cells, convert the motion of the basilar membrane into the electrical signals which are transmitted to the brain via the auditory nerve. This section presents how AIM models the adaptive thresholding as well as the approximations developed in this research to simplify the algorithms.

*3.5.1 Implementation in AIM.* After the logarithmic compression, the incident sound energy has passed through four transformations in the model. First, the middle ear (ear drum, malleus, incus and stapes) performed a band-pass filtering. Second, the energy caused the basilar membrane to resonate with an impulse response that is similar to that of the gammatone filter equation. Third, the energy was half-wave rectified and fourth logarithmically compressed.

The rectification and compression are actually only part of the mechanical-to-electrical transform accomplished by the hair cells in the organ of the corti. In addition to these two transformations, the hair cells also adapt to the motion of the basilar membrane to eliminate most of the ringing oscillations induced by the gammatone response of the basilar membrane. In AIM these transformations are known as two-dimensional adaptive thresholding[10] because it takes place in both the time and frequency domains. The following two sections discuss the adaptive thresholding in these two dimensions.

*3.5.1.1 Time Domain Adaptive Thresholding.* To understand the adaptive thresholding in the time domain, refer to the impulse response signal shown in Figure 22. The solid line segments represents the half-wave rectified, logarithmically-compressed motion at some point along the basilar membrane, or functionally, the energy that is stimulating the hair cells.

The hair cells which detect the motion of the basilar membrane are known to have a recovery time associated with their response. After they fire (transmit an electrical signal to the auditory nerve), they will not respond to further stimulation unless the new stimulation exceeds their current threshold level. In Figure 22, the threshold is shown as dashed lines. The threshold is a function of the hair cell's last firing level and time.

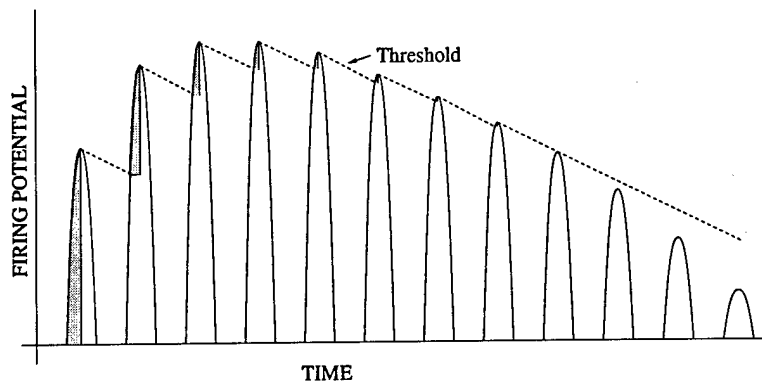


Figure 22: Time Domain Adaptive Thresholding (after [5])

In further study of Figure 22, assume that prior to the first rise in the input stimulus (the solid line) the hair cells have fully recovered from their previous stimulation. Therefore at the onset of the new stimulation, the hair cells will begin to fire in direct proportion to the intensity of the stimulation.

As the stimulus reaches its first peak and begins to fall, the hair cells cease to fire and enter their recovery period. The actual firing time of the cell is represented by the shaded portions of Figure 22. Initially, the stimulus required to cause them to fire again must exceed the previous peak, thus the firing level is the starting threshold level for the recovery period. Over time, the threshold decays as the hair cells recover from the previous firing. In the ear, the threshold is an exponential function with respect to the actual sound intensity. However, because the auditory system behaves logarithmically, the exponential decay becomes linear in the inner ear.

Once a non-zero threshold has been established, only signals which exceed the threshold will cause the hair cell nerves to fire. Even then, the firing of the hair cell is only proportional to the energy which is greater than the current threshold, not to the total energy of the stimulus. The result is a rapid damping of the ringing of the basilar membrane. In Figure 22, the shaded area under the input stimulation represents the resulting electrical signal sent to the auditory nerve after shifting each pulse down in order to align its base with zero potential.

While conceptually the process of adaptive thresholding may appear trivial, years of research and testing have resulted in the complex algorithm that is implemented by AIM.<sup>7</sup> The current

<sup>7</sup>The AIM code for the adaptive thresholding algorithm is found in the file `corti.c` which is located in the `model` subdirectory of the AIM 8.1 release. `Corti.c` models both the time and frequency domain thresholding.



algorithm parameterizes two separate rates for the decay of the threshold, as well as an onset rate. The onset rate allows for experimenters to adjust how rapidly the hair cell fires. Alternatively, the rise time can be thought of as how closely the hair cell's output follows an input pulse on the rising edge. The two decay rates work together through a feedback mechanism which makes the decay of the threshold slightly non-linear.

The algorithm receives as its input the half-wave rectified, logarithmically-compressed output of the gammatone filter bank. For each frequency channel, a threshold level is computed and stored. When a new pulse arrives on a channel, its magnitude is compared to the magnitude of the current decaying threshold. If the magnitude of the pulse is greater than the threshold, the difference is adjusted by the onset parameter and sent as output. In addition, the threshold is adjusted upward to the level of the new pulse. When the input pulse begins to fall off, the output of the thresholding algorithm immediately drops to zero because the input magnitude becomes less than the threshold.

In the ear, as with most real systems, there are no instantaneous changes in signal level. Therefore, the rapid return-to-zero of the output when the input signal falls below the threshold is unrealistic. The sharp fall-off is corrected by a low-pass filter which follows in the model.

After computing the output value, the current threshold is decayed. The new threshold value is computed based upon the current threshold, and the two decay parameters. The algorithm for the decay is discussed in Section 3.5.2 along with a discussion of the approximation to the algorithm.

*3.5.1.2 Frequency Domain Adaptive Thresholding.* In addition to adapting to signals in the time domain, each hair cell also interacts with its neighboring hair cells. Recall that the location of the hair cells along the cochlea determines the frequencies to which they will respond. Therefore, the interaction between hair cells causes a smearing effect across the frequency spectrum. The interaction between adjacent hair cells is the second dimension of the two-dimensional adaptive thresholding of AIM.

In AIM, the interaction between hair cells is modeled by allowing a change in threshold of one channel to propagate to and affect the threshold of its neighboring channels. The channel interaction is accomplished by multiplying the difference between the new input and the old threshold

by a leakage parameter. The result of the multiplication (if positive) is then added to the threshold of the neighboring channels.

It is important to note that the leakage parameter is a computed value based on the frequency separation between the adjacent channels. Because the separation between channels is inversely proportional to the number of channels, reducing the number of active channels increases their separation while reducing their interaction. Table 7 shows the magnitude of the lateral leakage parameter as a function of the number of channels in use. Table 7 was extracted directly from AIM by invoking AIM and varying the number of active channels.

Table 7: Lateral Leakage Parameters

Number of Channels	Lateral Leakage Multiplier
10	0.000183
20	0.000367
30	0.000550
40	0.000734
50	0.000917
60	0.001101
70	0.001284
80	0.001468
90	0.001651
100	0.001835
110	0.002018
120	0.002202
130	0.002385

Table 7 shows that a linear relationship exists between the number of channels and the lateral leakage constant, where the leakage coefficient can be computed by  $1.834 \times 10^{-5}$  times the number of channels. Perhaps more significant, the leakage parameter is a very small number, particularly when the number of channels is less than 60. As was illustrated in Figure 4 of Chapter 2, even with 64 channels, the effect of frequency thresholding was negligible.

For this research, 32 channels were chosen to be implemented. Currently, research on phoneme recognition uses only 18 channels[7, 25, 6]. Additionally, preliminary design calculations indicated that the conceptualized hardware architecture with an input data rate of 20,000 samples per second, could maintain real-time filtering on 32 channels when operated at approxi-

mately 28.5 MHz. The same hardware could process 44,000 samples per second (CD sample rate) when clocked at a modest 62.6 MHz.

Due to the minimal of effect frequency-domain adaptive thresholding, as compared to the overall magnitude of the NAP, frequency thresholding was not included in the architecture. If added, frequency adaptive thresholding would require three addition operations and one multiplication for each value passed through each channel. The increase in memory storage would be minimal since the thresholds are already being stored for the time-domain processing. However, the memory addressing scheme would become more complicated because of the need to access the data from more than one channel per operation.

**3.5.2 Approximation.** To better understand the approximation made to AIM's adaptive thresholding, first consider the original AIM C program code. The code fragment shown in Figure 23 was extracted from the file `corti.c` which is found in the `model` directory of the AIM release. For readability, the code in the figure has been modified to a pseudo-code format with line numbers added for reference. In addition, the code for frequency thresholding has been removed. Careful observation reveals two separate algorithms for computing the output value. The output is first computed in the block from line 4 through line 10. It is then computed a second time in lines 19 through 25. The second algorithm is the default for AIM. From comments that exist in the code, it is apparent that the first method was retained from an earlier release and remains as an option for backwards compatibility. Therefore, the first computation of the output was removed in the approximation. The lines eliminated were 4, 5, and 10.

The remaining code falls into three functional blocks: raising the current thresholds, decaying the thresholds, and generating the output. The following sub-sections explore the approximations that were derived for each of these functional blocks.

**3.5.2.1 Threshold Rise.** Beginning with the raising of the threshold in lines 1 through 12, note the outer loop which cycles from 0 to "times". When using the defaults in AIM, "times" is always set to 1, thus the loop cycles twice. Inside the loop the threshold is raised, and the parameter "rapid\_limit" is adjusted, affecting the decay rate. Both of these actions occur twice, making the decay rate slightly non-linear.

```

1  for( time=0 ; time < times ; time++ ) {
2      /* raise thresholds */
3      for( each channel ) {
4          if( time == 0 )
5              output = 0 ;
6          delta = input - microphonic;
7          if( delta > 0 ) {
8              microphonic = microphonic + (delta * rapid_rise) ;
9              rapid_limit = rapid_limit + (delta * fast_rise) ;
10             output = output + delta ;
11         }
12     }
13 }
14 /* decay potentials */
15 for( each channel ) {
16     microphonic =
17         microphonic - (microphonic-rapid_limit) * rapid_decay;
18     rapid_limit =
19         rapid_limit - (rapid_limit-absolute_limit) * fast_decay;
20 }
21 /* generate output */
22 for( each channel ) {
23     delta = input - microphonic ;
24     if( delta > 0 )
25         output = scale * delta / compensate ;
26     else
27         output = 0 ;
28 }

```

Figure 23: Adaptive Thresholding Code from corti.c

To simplify the process of raising the threshold, the loop was unrolled and the equations combined. For now, consider only the adjustment to the variable *microphonic* which is the current firing threshold. After the first pass through the loop, *microphonic* takes the form:

$$microphonic' = microphonic + rapid\_rise(input - microphonic) \quad (14)$$

On the second iteration, we replace all occurrences of *microphonic* with the newly computed *microphonic'* and simplify, yielding the expression:

$$microphonic'' = microphonic(1 - rapid\_rise)^2 + input((2 \times rapid\_rise) - rapid\_rise^2) \quad (15)$$

The parameter *rapid\_rise* is a channel-dependent parameter, i.e. each frequency channel has a unique value. Running a modified version of the AIM algorithm which displayed the values of these internal constants revealed that *rapid\_rise* took values from 0.21 to 0.25 for 100 Hz and 7 kHz channels respectively. Additionally, more of the channels had values of 0.25 or close to 0.25 as opposed to the lower values.

Since the variance of *rapid\_rise* was minimal, it was hypothesized (and later shown) that using the value 0.25 for all channels would have little effect on the algorithm. When the value 0.25 is put into the equation for *microphonic''* as defined previously, the equation simplifies to:

$$microphonic'' = (0.56 \times microphonic) + (0.43 \times input) \quad (16)$$

It was then noted that the average of these multiplier constants is approximately 0.5. Therefore, the approximation was extended so that the definition of *microphonic''* became:

$$microphonic'' = 0.5 \times (microphonic + input) \quad (17)$$

Equation 17 is equivalent to the Equation 15 if *rapid\_rise* is given the value 0.293 yet it can be implemented through a single addition with a shift right by one bit position of the result. Additionally, no ROM value needs to be stored for *rapid\_rise*.

The result of the approximation to *rapid\_rise* is that the rising edge of the thresholds for the neural firing will be slightly sharper than in AIM as seen in Figure 25. Additionally, because the same value for *rapid\_rise* (namely 0.293) is implemented for all channels, the firing thresholds of each channel will rise at the same rate. While giving each channel the same *rapid\_rise* parameter does remove some of the channel-to-channel differences that actually occur in the ear, the tests in Chapter 5 show that using the same *rapid\_rise* did not make a statistically significant change in the model when used for phoneme recognition.

Careful observation of the code fragment of Figure 23 shows one other parameter being computed in the threshold raising section: *rapid\_limit*. While computed in part when the threshold is raised, *rapid\_limit* is not used until the threshold decay section. Therefore, *rapid\_limit* is discussed in the following sub-section.

**3.5.2.2 Threshold Decay.** Immediately following the raising of the firing threshold due to a stimulation event, the threshold begins to decay. Thus as time passes, the inner hair cells recover their sensitivity. To model the hair cell's time-dependent recovery, AIM implements a time-dependent reduction to the thresholds of each channel. There are actually two different decay rates modeled in AIM simultaneously. The first is controlled by the parameter *rapid\_limit*, while the second is controlled by the parameter *rapid\_decay*.

The parameter *rapid\_limit* causes the rate of decay to be dependent upon the onset of the stimulation. Line 9 of the code segment in Figure 23 shows that *rapid\_limit* is set based on the difference between the threshold and the new stimulation. Numerically, *rapid\_limit* is initialized to the same value as the *absolute\_limit* parameter, which models the minimum input necessary to initiate a nerve firing event.

From its initial value, *rapid\_limit* is adjusted upward by multiplying the input, which exceeds the current threshold, by the parameter *fast\_rise* which is the key to the approximation. The parameter *fast\_rise* ranges from 0.00229, for a 6 kHz channel up to 0.0375 for a 100 Hz channel which equates to varying *rapid\_limit* by 0.2% to 3% of the amount that the input exceeds the current threshold.

The relatively small scale adjustment of *rapid\_limit* lead to the hypothesis that the combined effect of these two parameters could be eliminated and the value of the absolute limit could be used to model *rapid\_limit* as a constant. Allowing *rapid\_limit* to equal the absolute limit is consistent with AIM in that the absolute limit is the initial condition for *rapid\_limit*.

By choosing the absolute limit to model *rapid\_limit*, lines 9 and 17 are removed from the code of Figure 23. After removing *fast\_rise*, substituting *absolute\_limit* for *rapid\_limit*, and performing minor algebraic manipulations, line 16 can be expressed as:

$$microphonic = microphonic(1 - rapid\_decay) + (absolute\_limit \times rapid\_decay) \quad (18)$$

AIM was employed to generate the needed values for *rapid\_decay* which are required by Equation 18. As expected, these tests revealed *rapid\_decay* to be channel-dependent. Specifically, it ranges from a low value of 0.001428 for the 100 Hz channel to a high value of 0.027047 in the 6 kHz channel. Experimenting with AIM revealed that the generation of the neural activity

pattern (NAP) is much more influenced by *rapid\_decay* than all of those previously discussed. The dependency of the NAP on *rapid\_decay* can be understood by realizing that these values are computed to closely follow the natural decay rate of the gammatone impulse response. It is the slope, as shown in Figure 22, that determines which events will cause a nerve firing event. Therefore, care had to be taken in how the values for *rapid\_decay* were handled.

Because the architecture being considered was a fixed-point integer architecture, fractional values had to be converted into a fixed-point form. When converting *rapid\_decay* into a binary fixed-point notation, it was noted that if the decimal point is shifted by 16 bit positions (equivalent to multiplying the value by  $2^{16}$ , the largest possible value in the proposed system), the resulting integers occupied, at most, 12 bits. To reduce hardware costs and processing time, the size of the constants were rounded to 12 bits. A 12-bit fixed-point integer approximation was then coded into AIM and validated.

One final approximation was made in the adaptive thresholding section of AIM. The value for *absolute\_limit*, the lower limit of the threshold, is coded in AIM as 1397.94. Because of the simplicity of the binary form of 1024, and the ease with which it can be multiplied, experiments were run in which *absolute\_limit* was approximated as 1024. While the magnitude of the NAP was slightly larger when using 1024, phoneme recognition experiments produced nearly identical results as those when *absolute\_limit* was its original 1397.94.<sup>8</sup> Therefore, 1024 was adopted as a valid approximation.

**3.5.2.3 Output Generation.** The final step in the adaptive thresholding process is generating the actual output by determining the difference between the current threshold (which has been decayed) and the input. If the input is less than the threshold (*microphonic* in the code), the output is set to zero. Otherwise the output is a positive value based on the difference between the signal input and the threshold.

Rather than simply transmitting the difference between the input and the threshold, AIM multiplies the difference by a compensation factor. These compensation coefficients (one per chan-

---

<sup>8</sup>Accuracy for these approximations was measured by using AIM as a front end processor for phoneme recognition software explained in more detail in Section 5.2. During the testing, candidate approximations were coded into AIM and run through a battery of recognition experiments. The results from these experiments were used to compare the effectiveness of the approximations to the results produced by the original AIM code.

nel) are related to the *rapid\_decay* parameters. Two techniques were attempted to approximate the compensation factors of AIM.

The first approximation tested used the values of *rapid\_decay* indexed in reverse order and multiplied by 2048. The multiplication by 2048 approximates the actual relationship between *rapid\_decay* and the compensation factor which actually varies from channel to channel. The value 2048 was chosen not as an average, but rather for speed. The attractiveness of this technique is the fact that the data is already stored and the multiplication is simply a shift of 11 bits to the left.

While the first approximation for *rapid\_decay* could have saved ROM space in the architecture, it did not survive the phoneme recognition test and subsequently was not used. Additionally, the hardware realization of the first approximation would have complicated the ROM addressing technique by requiring a more random access to the *rapid\_decay* memory.

The second approximation attempt came from a study of the compensation constants generated by AIM. Testing revealed that the values for the compensation factor ranged from a high of 66.85 for the 100 Hz channel, to a low of 3.44 for the 6 kHz channel. Therefore, a logical approximation was an 8-bit constant with 6 bits of integer and 2 bits of fixed-point fraction. The fraction was rounded to the nearest multiple of 0.25 to allow the most accurate storage of the compensation factor with two bits of fraction.

By limiting the compensation factor to between 0.0 and 63.75, the approximation closely replicates (within rounding to 0.25) the AIM constants to filters as low as 121 Hz. Restricting accurate compensation to channels above 121 Hz is not seen as a hindrance to the model since current phoneme recognition experiments do not use channels below 350 Hz[25].

Figure 24 illustrates the new approximated algorithm inserted into AIM for testing. As was the case with Figure 23, the code of Figure 24 was simplified by abstraction for readability.

To test the adaptive thresholding approximation, an impulse was used as stimulus for AIM running both the original, and the new code. The output from these two runs was subtracted and plotted in the form of a NAP. Figure 25 shows the resulting difference between the given approximation and the original AIM thresholding algorithm.



```

1  for ( each channel ){
2      /* Decay the threshold */
3      microphonic =
          microphonic * (1-rapid_decay) + (absolute_limit * rapid_decay)
4      delta = input - microphonic;
5      if (delta > 0) {
6          /* Generate Output */
7          output = delta * compensate;
8          /* Raise Threshold */
9          microphonic = (microphonic + input)/2;
10     }
11     else
12         output = 0;
13 }

```

Figure 24: Approximated Adaptive Thresholding

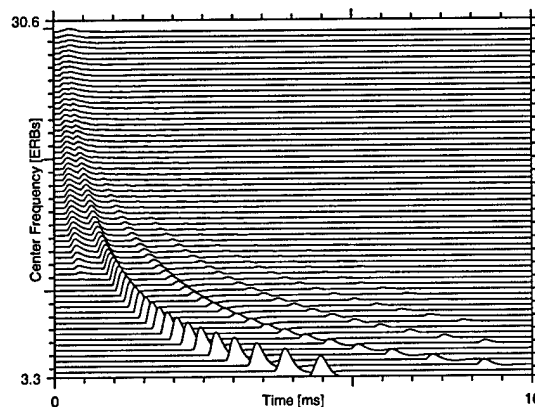


Figure 25: Difference Between AIM NAP and Approximated NAP

In generating Figure 25, the output for the case of the approximation method was scaled by 0.7 to remove the magnitude difference when approximating *absolute\_limit* as 1024. After removing the scaling effect, it is easy to see that the rising edges of the nerve firings differ the most. For comparison, the baseline NAP for Figure 25 is the NAP of Figure 3. What is perhaps most important to observe from Figure 25 is that the number of nerve firings and their locations in time are consistent with those produced by AIM.

### 3.6 Integration Filtering

The final stage of processing in the production of a neural activity pattern is a low-pass integration filter which smoothes the output of the adaptive thresholding to remove the finite discontinuity that results from the abrupt turn-off. Recall that in the thresholding stage (Section 3.5.1.1), an output was generated whenever the input exceeded the current threshold. However, as soon as the input dropped below the current firing threshold, the output was immediately forced to zero producing an undesirable discontinuity in the output which does not exist physiologically.

The integration filter is actually a part of the neural encoding stage since it is required to properly shape the output to more closely match the output of the hair cells. For clarity, the discussion of the integration filtering has been removed from the neural encoding and is presented in this section.

**3.6.1 Implementation in AIM.** To eliminate the rapid fall time of the signal, AIM introduces a low-pass (integrating) filter.<sup>9</sup> One integration filter is common to all channels; there are no channel specific constants. However, state information must be maintained on a channel basis. Mathematically, the filter is a first-order IIR type filter implemented in AIM by the difference equation:

$$Y[n] = Y[n-1] + K(X[n-1] - Y[n-1]) \quad (19)$$

where  $K$  is the constant 0.313356,  $X$  is the input, and  $Y$  is the output. Operationally, the data is recursed through Equation 19 twice to produce a second-order filter. The frequency response of the second order filter is shown in Figure 26.

Through algebraic manipulations, the characteristic difference equation for the first-order filter can be converted to a more standard form represented by:

$$Y[n] = KX[n-1] + CY[n-1] \quad (20)$$

In Equation 20,  $K$  is 0.313356 as before, and  $C$  is  $(1-K)$  or 0.686644. The same expression may also be converted into its second order form, representing the implementation of the two stages of

---

<sup>9</sup>The C code for the integration filter module can be found in the `model` directory of the AIM distribution under the filename `integrate.c`.

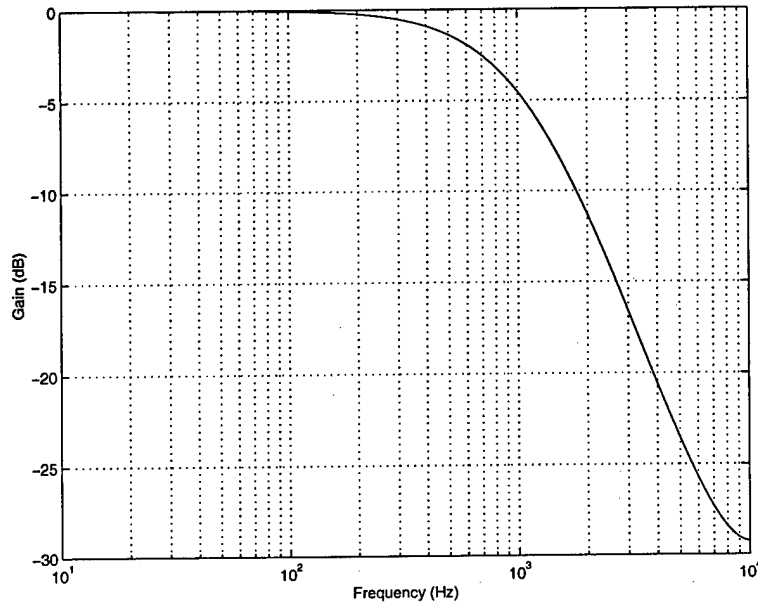


Figure 26: Frequency Response of AIM's Integration Filter

filters. However, it is worth pointing out that implementing the combined second-order equation directly increases the number of required constants, and results in smaller constants which are more susceptible to rounding errors.

**3.6.2 Approximation.** Because of the relative simplicity of the integration filter implemented in AIM, it was not replaced. A size optimization on the filter constants was performed. Before the optimization could begin, it was necessary to decide whether to implement the recursive first-order filter as in AIM, or to combine the stages to form a second-order filter. The first-order filter was selected because of the number and the size of the constants required by the second-order filter.

The word size optimization was performed by manipulating the binary form of the two constants  $K$  and  $C$  that were discussed in the previous section. Specifically, these constants (rounded to sixteen bits of fraction) are:

$$C = 0.0101000000111000_2 \quad K = 0.1010111111000111_2 \quad (21)$$

Note that the bit patterns of these two values in the 5<sup>th</sup> through 8<sup>th</sup> positions to the right of the decimal point warrant rounding the values to 4 bits of fraction. As a matter of fact, 99.72% of  $C$  is captured in these first four bits of fraction. Similarly, if  $K$  is rounded to 0.1011<sub>2</sub>, it is only 0.12% above the AIM value. Figure 27, which is a plot of the difference between the frequency response of the exact filter and the approximate filter, illustrates that the minor changes in the coefficient values have a negligible effect on the performance of the filter.

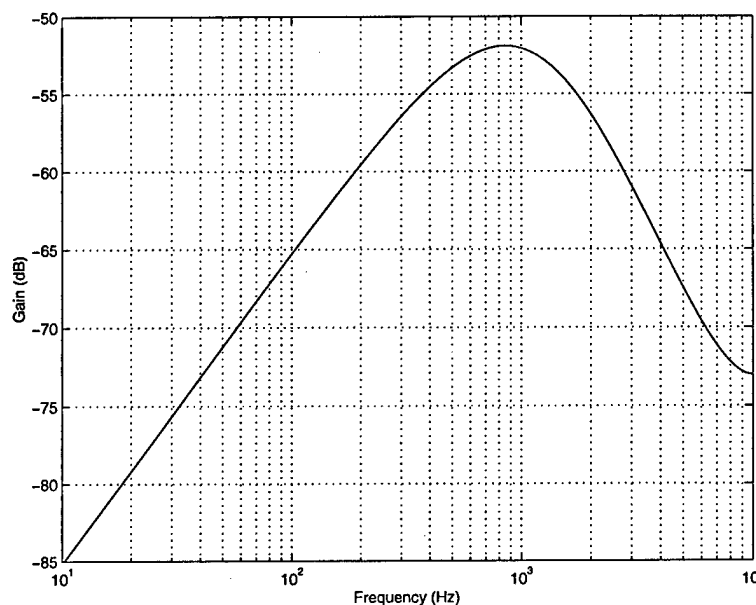


Figure 27: Difference Between Exact and Approximated Integration Filters

Figure 27 was generated by computing the complex frequency response of the AIM integration filter and the bit-truncated integration filter just discussed. The frequency responses (in complex form) were then subtracted and the resulting magnitude was plotted in dB form as shown.

To fully appreciate the insignificance of the difference between the AIM filter and the approximated filter, refer to Figures 26 and 27. Note that the peak difference shown in Figure 27 corresponds to the -3dB point in Figure 26. The maximum difference between the AIM filter and the approximate filter is -52dB which corresponds to a 0.251% difference in magnitude.

Thus, the resulting filter is shown to be nearly identical in performance to the original AIM filter, but with the advantage of requiring a much smaller binary representation. The advantage of the smaller representation is manifested when constructing the actual hardware in two primary

modes. First, the size of the constants require smaller storage. Second, the actual multiplication is one-fourth of that required for a 16-bit multiplication.

### 3.7 *Integrated AIM Approximation*

The complete approximation model consists of five separate components: middle-ear IIR filter, gammatone filter bank, logarithmic compression unit, neural encoding, and a low-pass (integration) filter. These five components were all coded using the C programming language as modules which functioned as callable replacements for AIM's modules.

To illustrate the functioning of the completed approximation model, Figures 28, 29, 30, and 31 are included. Figure 28 shows the default AIM NAP generated for the spoken word "hat". For comparison, Figure 29 is the same spoken word as processed by the approximation code. Similarly, Figures 30 and 31 illustrate the NAPs generated for a 1 kHz sine wave as generated by AIM and the approximated code respectively.

In comparing Figure 28 to Figure 29 and Figure 30 to Figure 31, some observations can be noted. First, the overall structure of the NAP is preserved in the approximation. Secondly, however, some differences exist in the finer details. For example, the approximation appears to be lacking some pulses in the range of 0 to 120 ms, particularly in the low frequencies. Conversely, in the range of 580 to 600 ms, the approximation appears to have additional details not present in the AIM NAP. Additionally, the approximate NAP has been magnitude scaled to account for the difference in the absolute threshold. By removing the magnitude difference (a constant multiplier), the structural differences between the NAPs is more easily observed.

### 3.8 *Summary*

In this chapter each stage of AIM's default functional model was examined. For each AIM stage the underlying algorithms were presented and approximations were developed which simplify the hardware implementation of these algorithms. Additionally, spectral analysis and phoneme recognition tests were conducted to verify that the approximations did not adversely affect the overall functioning of the system.

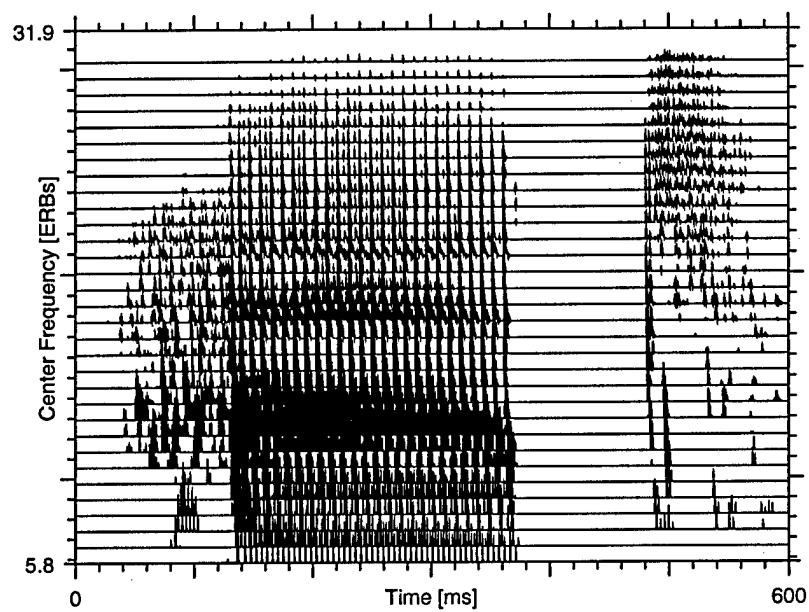


Figure 28: NAP of "HAT" Generated by AIM

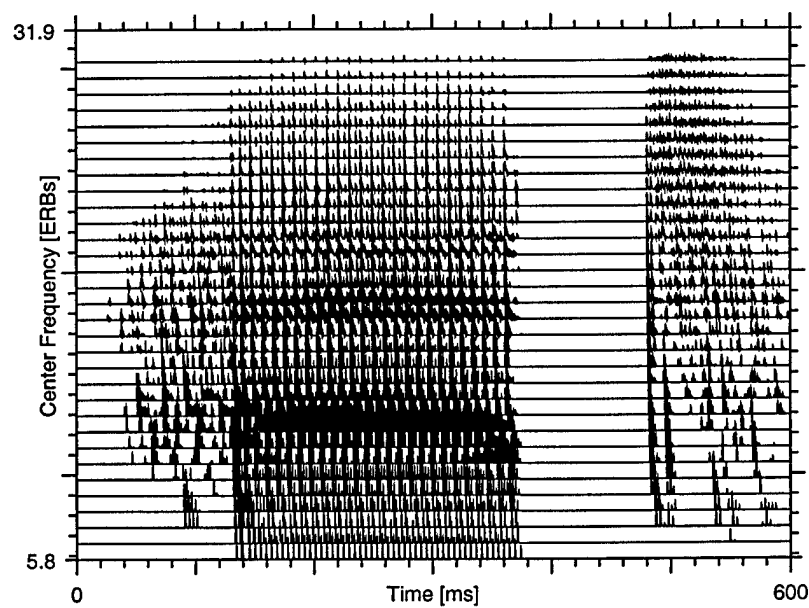


Figure 29: NAP of "HAT" Generated by Approximated Code

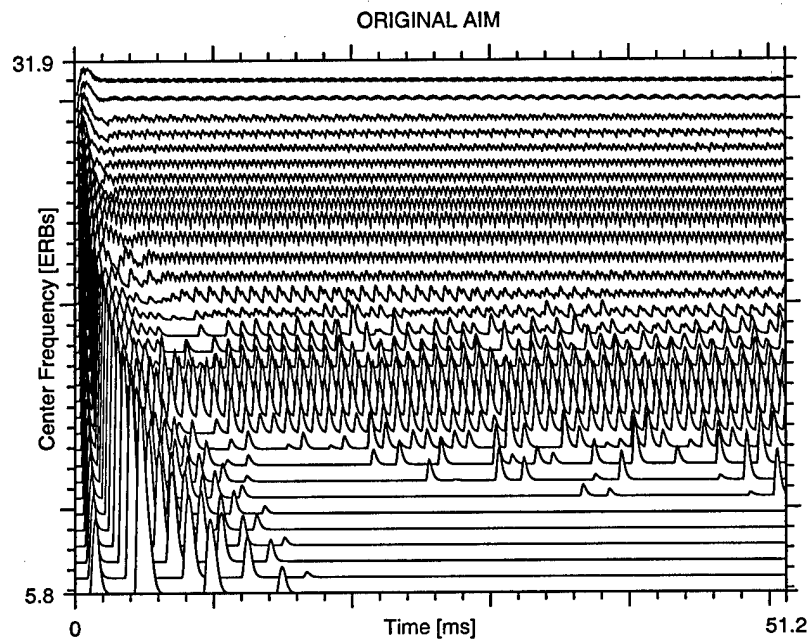


Figure 30: NAP of "1kHz Sinusoid" Generated by AIM

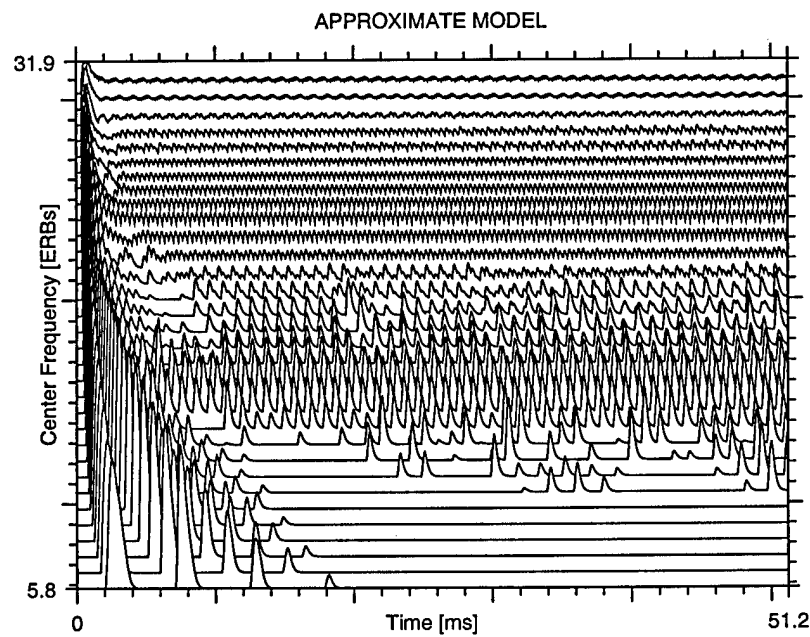


Figure 31: NAP of "1kHz Sinusoid" Generated by Approximated Code

The verification of each approximation included the coding of the approximation into AIM compatible modules. In several cases VHDL and Matlab modeling were used to complete the verification. VHDL modeling was used to accurately model bit-level implementations of the algorithms, allowing for accurate assessment of word sizing and truncation errors. Matlab modeling assisted in the development of filter equations, and was relied upon for the post-processing of data extracted from both the AIM and the VHDL tests for spectral analysis. The result of this portion of the research was a set of algorithms and equations which will accurately and efficiently imitate the functioning of AIM but will simplify a hardware implementation.

With the development of the algorithms complete, the next step was to design a hardware architecture on which the algorithms could be efficiently executed. Chapter 4 is devoted to a discussion on the design of such an architecture.



## *IV. Hardware Implementation*

### *4.1 Introduction*

Following the development of the approximation algorithms for AIM, the next step was to design a system in which the algorithms could be efficiently executed. The goal of the design phase of the research was to demonstrate the feasibility of designing a single-chip system capable of processing as many as 32 frequency channels in real time. The present chapter documents the resultant architectural design.

The architecture under discussion has been modeled to the gate level using the VHSIC<sup>1</sup> Hardware Description Language (VHDL). The purpose of the VHDL model was two-fold: to show that the proposed architecture could be constructed using current VLSI technologies, and to demonstrate that the algorithms of the modified AIM were correctly implemented.

Architecturally it is shown that the proposed system is, in part, a single instruction, multiple data (SIMD) architecture. Additionally, the architecture is pipelined to allow for overlapping execution where possible. A third characteristic of the design is that it performs multi-rate operation. Data enters the system at a rate that is lower than the data is processed internally.

As in the preceding chapters, this chapter follows the general organizational flow of AIM with minor exceptions. Rather than discussing each of the modules of AIM individually, some of the modules have been combined. Specifically, the processing of the outer/middle-ear filter (OMF) and the all-pole gammatone filterbank (APGF) were combined into the first stage. The second processing stage accomplishes the half-wave rectification and the logarithmic compression. Finally, in the third processing stage the neural encoding was combined with the integration filter. As will be shown, regrouping some modules simplified the hardware design and allowed for maximum re-use of functional units. The three stages are discussed in the following sections. The first stage to be considered will be the filtering stage.

---

<sup>1</sup>Very High-Speed Integrated Circuit

## 4.2 Filtering

**4.2.1 Algorithm.** The first stage of the proposed hardware model combines the first two stages of AIM, namely the outer/middle-ear filter (OMF) and the gammatone filterbank. The corresponding approximation filters are the IIR filter, and the all-pole gammatone filterbank (APGF). There were three reasons for combining these two units: hardware utilization, similarity in processing requirements, and data precision.

First, as data enters the model, it is processed one time through the OMF. In comparison, the filterbank must process each data value four times for each filter in the bank to achieve the required filter order. As a result, any functional hardware dedicated solely to the OMF would be under-utilized while waiting for the filterbank to complete its processing. Because of the data dependency between the OMF and APGF, the execution of these algorithms remains mutually exclusive; only one of the two can be in execution at any given time. Therefore re-using functional units is logical.

Secondly, the IIR filter requires six multiplications and five additions while the gammatone filter requires three multiplications and two additions. Since the number of multiplication operations in each is a multiple of three, the two filters map easily onto the same underlying parallel hardware core. Since the multiplications require several clock cycles to complete, the difference in the number of additions can easily be hidden by the use of multiplier output latches and a single adder operating as an accumulator.

Finally, by combining the two filter stages into one functional unit, the internal precision of the data is improved. Specifically, the number of fractional bits in the internal processing, as well as the storage, can be increased without the additional cost of a wider data path between the stages. Thus the input and the output of the combined unit may remain integer, while all filtering can be accomplished with the added precision of fixed-point arithmetic.

The mapping of both filtering algorithms onto the same hardware core was not the only implementation considered. An option where the filtering of the OMF and the APGF were combined into a single equation was also considered. In order to combine the OMF and APGF, each filter in the APGF filterbank is scaled according to the magnitude of the OMF response at the center frequency of the particular APGF filter. The scaling, accomplished by weighting the input coeffi-

cient of the all-pole gammatone filters, results in a complete removal of the OMF without adding computation to the APGF.

While initially attractive, the combined algorithm concept was not implemented because the technique would scale the entire frequency response envelope of the gammatone filter by the same factor. In particular, a constant scaling factor would artificially raise the response of APGF in the low frequencies where the OMF actually imposes a loss. Therefore, simply scaling the envelope of the gammatone filter would result in a distorted frequency response of the combined OMF/APGF filters, particularly at the low frequencies.

Having eliminated the proposition of combining the filter equations, an architecture was designed which supported the direct computation of both filter equations using shared functional and memory units. A block diagram of the filter stage architecture is presented in Figure 32. Recall the equations for the IIR filter and the gammatone filter discussed in Chapter 3:

$$Y_{[n]} = A_1 X_{[n-1]} + A_2 X_{[n-2]} + A_3 X_{[n-3]} + B_1 Y_{[n-1]} + B_2 Y_{[n-2]} + B_3 Y_{[n-3]} \quad (22)$$

$$Y_{[n]} = AX_{[n-1]} + BY_{[n-1]} + CY_{[n-2]} \quad (23)$$

The discussion of Figure 32 is broken down into the following three subsections: multipliers, memory, and control components.

Functionally, when data enters the system it is stored in the  $X - IN$  register. The previous two inputs are then recalled from  $RAM1$  and  $RAM2$ . All three input values then enter the multipliers where they are multiplied by the first three coefficients of the OMF equation. The results of the multiplications are latched and summed via the accumulator structure consisting of  $MUX1$ ,  $MUX2$ ,  $ADDER$ , and  $OUTPUTDATA LATCH$ .

While the accumulating is taking place, the previous output of the OMF is recalled from the  $Y(-1)$  latch while the two outputs prior to the previous are recalled from  $RAM1$  and  $RAM2$ . The three previous outputs then enter the multipliers where they are multiplied by the corresponding coefficients of the OMF equation. When completed, the system waits while these scaled outputs are accumulated with the inputs to complete the OMF processing. Finally, the accumulated value is written into the  $Y(-1)$  latch which becomes the input to the APGF.

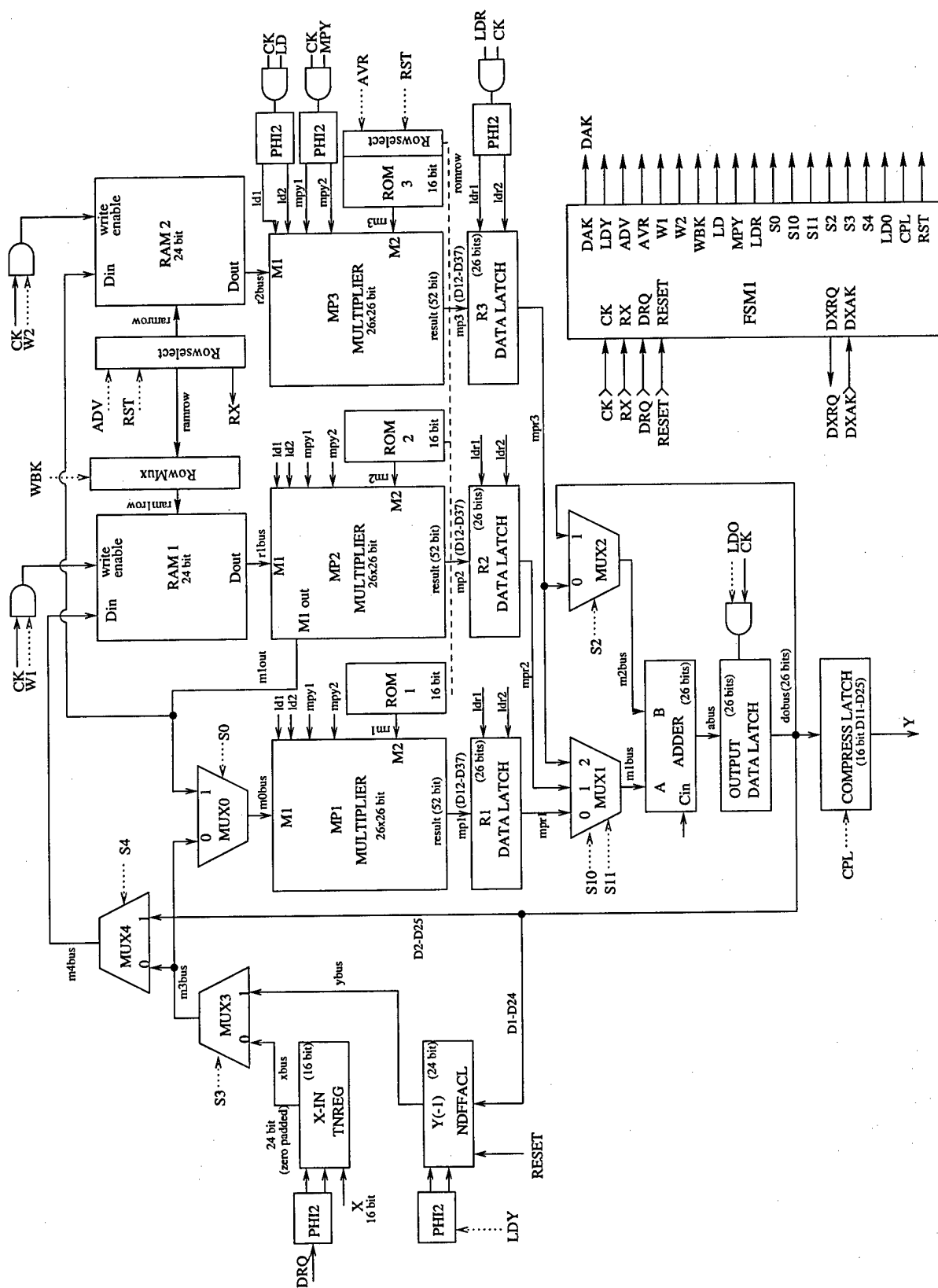


Figure 32: Filtering Stage Block Diagram

The processing of the APGF is similar to the OMF except that only one pass through the multipliers is required. During the multiply phase, the value stored in  $Y(-1)$  is used as the X input. The two previous outputs of the APGF filter are recalled from memory allowing the complete APGF equation to be computed.

Following is a discussion of the details of selected portions of the design. The hardware discussion begins with the multipliers.

**4.2.2 Multipliers.** The multiplication operations were the starting point for the design due to their high relative cost in area and computational time compared to the other required operations. By choosing to implement three multipliers in parallel, one-half of the IIR filter multiplications, or all of the multiplications required by the all-pole gammatone may be computed simultaneously.

The multipliers chosen for the architecture were recursive octal-encoded Booth multipliers. While the fully combinational form of the Booth multiplier can be configured to execute in a single machine cycle, the recursive technique was chosen for two primary reasons. First, the recursive form of the Booth multiplier is smaller than the combinational form because of the re-use of a single adder. To perform the same multiplication using a combinational multiplier would require eight 25-bit adders compared to the one 26-bit adder of the sequential multiplier. Analysis proved that real time operation for a bank of 32 channels could be achieved using the slower recursive method. (See Section 5.3 for a summary of the speed analysis.)

The internal design of the multipliers was accomplished using a bit-slice technique. Each slice contains an adder/subtractor, an input latch for the multiplicand, multiplexers, and a pair of data latches for holding the result. Initially, the multiplier is loaded into one-half of the output register. As the multiplier is shifted for the octal Booth's algorithm, the result is shifted into the same register. Upon completion, the multiplier value is fully shifted out, and the result remains in the latch pair.

The multiplier slice does not contain any decode logic for the Booth algorithm. The decoding is done by a small finite state machine attached to the array of slice cells. Operating from a single clock, the state machine controls the adders, multiplexers and latches. The multiplier performs one Booth's cycle for each clock period it receives. In the design, the multiplier state machine does not

keep track of the number of cycles to perform. Rather, the cycle counting is performed external to the multiplier slices so that the same cells can be configured to operate on different word sizes if needed.

The critical component limiting the operational speed of the multiplier is the adder. Here ripple-carry adders were chosen for their small size in comparison to other adder designs such as the carry select or carry look-ahead adders. The required multiplications of the filter bank are 24 by 16 bits. Therefore, in order to ensure that the recursive Booth algorithm computes negative numbers correctly, the minimum adder size is 26-bits. The additional bits are required to allow the Booth algorithm to perform a conditional multiply by two before the add/subtract and to preserve the sign in all cases.

If we allow a conservative 1 ns delay<sup>2</sup> per carry stage, the 26-bit ripple carry adder design would have a maximum delay of 26 ns, thus limiting the design to speeds under 38.4 MHz. As will be shown in Chapter 5, a clocking speed of only 27.5 MHz is required to sustain real-time operation in a 32 channel system. Therefore the ripple-carry adders will provide sufficient execution speed.

Because the multiplier coefficients for both the IIR and the APGF are 16 bit, only 8 clock cycles are needed to complete a multiplication using the octal Booth algorithm. However, recall from Chapter 4 that the coefficients for the IIR differ in precision from those of the APGF. Specifically, the IIR has 3 bits of integer and 13 bits of fraction while the APGF has 2 bits of integer and 14 bits of fraction. The difference in precision requires that an adjustment be made to re-align the data after the multipliers. As indicated by the bit-field labels on *dobus* of Figure 32, the data re-alignment is accomplished in the routing of the data during write-back to storage.

**4.2.3 Memory.** Following the design of the multipliers, the next issue to be addressed is the memory requirements of the filtering stage. Looking again at Equations 22 and 23, we see that storage for six internal states (24-bit words) are required for the IIR filter. Additionally, three 24-bit storage cells are required in each stage of recursion for each filter in the filterbank. Therefore, a system with 32 filter channels would require 390 memory storage cells of 24 bits. However, by

---

<sup>2</sup>Spice simulations for different adder cells considered for the design were consistently less than 1 ns when using 0.8  $\mu$ m technology data. The delays will be even shorter if the design is migrated to a smaller technology.

translating the first recursion of the APGF in time we may take advantage of the spatial locality of the data and reduce the storage requirement to 262 words of 24-bits.

The reduction in required memory is accomplished by eliminating the need to store a history of inputs for the APGF bank. The need for a history of input values is eliminated by translating the first recursion of the APGF one sample forward in time. The forward translation is performed by allowing  $X_{[n-1]}$  to become  $X_{[n]}$  in Equation 23. Therefore, the present output from the IIR filter becomes the input to the filterbank during the same sample period that it is computed.

One side effect of time translation is seen as the first output from the filterbank arrives one sample earlier than it would otherwise. The resulting shift in time does not change the shape or spectra of the filter's response to an input. The only actual change in the data stream is that the output from each channel will have one less leading zero before the filter begins to respond to an input. Since all of the channels experience the same time slippage, there is no phase shift between channels.

Continuing with the memory reduction effort, note that in computing the APGF the previous output ( $Y_{[n-1]}$ ) is recalled from memory to be multiplied by the  $B$  coefficient. Since the fourth-order APGF is computed by four passes through the same filter equation, the previous output of each recursion becomes the input to the next. Therefore,  $Y_{[n-1]}$  is needed as the input  $X_{[n-1]}$  on the next pass of the filter. By exploiting the fact that  $Y_{[n-1]}$  is currently held in a multiplier, we can eliminate the need for separate storage and/or retrieval of the value from memory. To take advantage of the location of the data, a data path (*mlout* in Figure 32) was added to allow the data to propagate directly from one multiplier to the other.

Since the optimized architecture has only two banks of read-write memory (RWM), two latches were added to the design to accommodate the six storage locations required by the IIR filter. These latches are used to hold the present input to the IIR filter and the newly computed IIR filter output. The added latches work to the advantage of the overall system by simplifying the fetching of the IIR filter's output for the start of each filter in the APGF bank.

The memory in the proposed architecture has been designed to simplify its access. Specifically, the filter states are stored in two banks of read-write memory (RWM) with their addresses aligned. Therefore, a single memory address register ("Rowselect" in Figure 32) can be used to

access both banks simultaneously. Similarly, the three ROM coefficients for any given filter are likewise stored in three banks of address-aligned ROM. The ROM does require an address register separate from the RWM because the same coefficients are used for each recursion of any given filter. Therefore, the ROM banks only need to have one-fourth of the number of memory locations as the RWM banks.

There is an exception to the RWM alignment. In order to reduce the overall time required to execute the filterbank, an overlap of operations was incorporated. After the first recursion of a given filter completes its multiplication phase, the multipliers are immediately loaded for the second recursion. While the second multiplication is being initiated, the products from the first are being summed. Therefore, before the sum of the first products is completed and ready to be written back to memory, the RWM address has been advanced. To allow the memory write-back to occur to the previous address, the "Rowmux" shown in Figure 32 was added. The "Rowmux" multiplexer allows the memory write-back to occur either at the current or previous address. The actual destination is selected by the controlling state machine discussed in Section 4.2.4.

In order to automate the generation of the ROM, AIM was modified so that it would automatically generate files which contained the data for the ROMs. All of the values are represented as integers in these files. That is, all fractions have been left-shifted by the appropriate number of bits to align the decimal point after the number. The ROM data file is in a form that allows it to be used as input to the VHDL model directly, or to be used to generate a VLSI ROM layout.

Additionally, a C program was written to generate a ROM layout. (See Appendix D.) The ROM generation program requires a file of integers as input and generates a file which is the VLSI layout of a programmed ROM. The resulting layout file may be used as a sub-cell in an actual system layout.

In order to access the memory (both RWM and ROM), circuitry must be included which enables one row of cells for reading or writing. A chain of data latches was chosen for the row selection. Each latch is initialized to '0' upon reset except for the first latch which is initially set to '1'. Upon each advance command from the controller, the '1' is transferred to the next cell in the chain and the cell giving up the '1' returns to a '0' state. There is one latch for each row in the memory array; the row corresponding to the latch which holds the '1' is the selected row.



A bucket brigade method was chosen to eliminate address decoders that would be required if a binary counter were used. In addition, the bucket brigade method consumes considerably less power than a full decoder because of the reduction in the number of bit transitions per clock cycle. The bucket brigade addressing technique could fail if an external force (such as a power surge) caused more than one latch to be loaded with a '1'. The effects of such a condition would be minimal because the system automatically resets these registers upon the receipt of each new data value.

**4.2.4 Control.** The control of the filter-bank subsystem resides within a finite state machine pictured in Figure 33. The finite state machine provides all of the signals to the multiplexers, adders, multipliers, and memory in order to implement the algorithms. The design of the finite state machine began with the drafting of a conceptual architecture. The modified AIM algorithms were then mapped onto the draft architecture and logic was added as necessary until all of the algorithms could be implemented. The resulting filter architecture is shown in Figure 32.

Next, a state table was generated in which all of the needed control signals were clearly identified as well as an initial assignment of states. Table 8 identifies the value of each control signal of Figure 32 in each machine state. In the first state table, every step of every iteration was delineated for clarity. Once completed, the table was examined for redundancies and loops were inserted which reduced the number of required states.

Notationally, the dots in the state table represent "don't-care" values. In the "Next" column, signal names that appear with the next state are entry conditions to that state. For example, to exit from state 21 the signal named *MP8* (which counts the 8 cycles of the multiplier) must be asserted and the signal *ST4* (which counts the 4 recursions of the filter) is checked. When *ST4* is asserted, the next state will be 22 signifying the completion of the filter channel; otherwise the next state will be 17 where another recursion is started. The equations derived for each of the output signals are tabulated in Appendix B.

The signals *DRQ* (data request) and *DAK* (data acknowledge) are not included in Table 8. *DRQ* is the asynchronous data request from the sending unit, while *DAK* is a signal returned to the sender to acknowledge receipt of the data. When the sending process has data, it first checks to

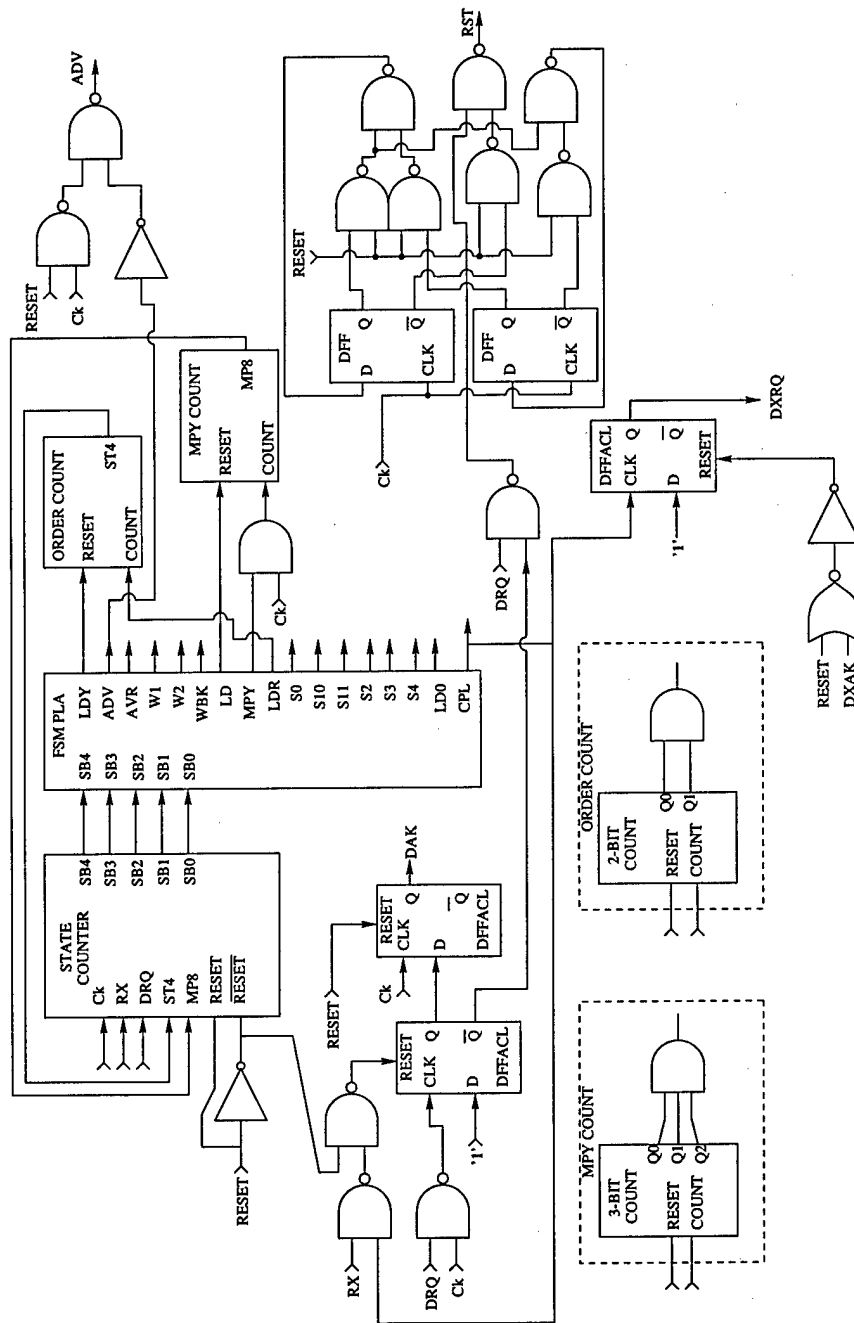


Figure 33: Filtering Finite State Machine

Table 8: Filter Section State Table

State	ldy	rst	adv	avr	w1	w2	wbk	ld	mpy	ldr	s0	s10	s11	s2	s3	s4	ldo	cpl	Next
0	0	0	0	0	0	0	.	0	0	0	.	.	.	.	.	.	0	0	1:DRQ
1	0	1	0	0	0	0	.	0	0	0	.	.	.	.	.	.	0	0	2
2	0	0	0	0	0	0	.	1	0	0	0	.	.	.	0	.	0	0	3
3	0	0	0	0	1	1	0	0	1	0	.	.	.	.	0	0	0	0	4
4	0	0	0	0	0	0	.	0	1	0	.	.	.	.	.	.	0	0	5:MP8
5	0	0	1	1	0	0	.	0	0	1	.	.	.	.	.	.	0	0	6
6	0	0	0	0	0	0	.	1	0	0	0	0	0	0	1	.	1	0	7
7	0	0	0	0	0	0	.	0	1	0	.	1	0	1	.	.	1	0	8
8	0	0	0	0	0	0	.	0	1	0	.	.	.	.	.	.	0	0	9:MP8
9	0	0	0	0	0	0	.	0	0	1	.	.	.	.	.	.	0	0	10
10	0	0	0	0	0	0	.	0	0	0	.	0	0	1	.	.	1	0	11
11	0	0	0	0	0	0	.	0	0	0	.	1	0	1	.	.	1	0	12
12	0	0	0	0	0	0	.	0	0	0	.	0	1	1	.	.	1	0	13
13	1	0	1	1	1	1	0	0	0	0	.	.	.	.	1	0	0	0	14
14	0	0	0	0	0	0	.	1	0	0	0	.	.	.	1	.	0	0	15
15	0	0	0	0	0	1	.	0	1	0	.	.	.	.	.	.	0	0	16
16	0	0	0	0	0	0	.	0	1	0	.	.	.	.	.	.	0	0	17:MP8
17	0	0	1	0	0	0	.	0	0	1	.	.	.	.	.	.	0	0	18
18	0	0	0	0	0	0	.	1	0	0	1	0	0	0	.	.	1	0	19
19	0	0	0	0	0	1	.	0	0	0	.	1	0	1	.	.	1	0	20
20	0	0	0	0	1	0	1	0	1	0	.	.	.	.	.	1	0	0	21
21	0	0	0	0	0	0	.	0	1	0	.	.	.	.	.	.	0	0	17:mp8 ST4 22:MP8 ST4
22	0	0	1	1	0	0	.	0	0	1	.	.	.	.	.	.	0	0	23
23	0	0	0	0	0	0	.	1	0	0	0	0	0	1	.	.	1	0	24
24	0	0	0	0	0	1	.	0	1	0	.	1	0	1	.	.	1	0	25
25	0	0	0	0	1	0	1	0	1	0	.	.	.	.	.	1	0	1	16:MAX 0:RX

see that *DAK* is low. If *DAK* is low, the sender places data on the input of the filterbank and sets *DRQ*.

Upon receiving the asserted *DRQ*, the filterbank leaves its idle state (state zero) and asserts *DAK*. *DAK* will remain asserted until all filters in the bank have completed the processing of the newly arrived data. When the processing is complete (signaled internally by advancing the RWM address beyond the last actual memory address), *DAK* is un-asserted and the filterbank returns to idle, pending the arrival of the next data sample.

Four processing loops are evident in Table 8. All of these depend upon the signal *MPY8*. The signal *MPY8* is generated by a counter which keeps track of how many clock cycles are issued to the multiplier. Although the multiplier, as previously discussed, is physically  $26 \times 26$  bits, the ROM constants are only 16 bits. Therefore, only 8 clock cycles are needed for each of the multiplications. As indicated in Table 8, the state machine will remain in states 4, 8, 16, and 21 pending the completion of an 8-cycle multiplication.

The first two multiplication states (states 4 and 8) compute the outer/middle IIR filter. The third multiplication state (state 16), computes the first recursion of the APGF, while state 21 computes the second through fourth recursions of the APGF. A unique multiply state was necessary for

the first recursion of the APGF since its input is the output from the IIR filter. The input to each of the remaining gammatone recursions is the output from the previous recursion.

State 21 has a second loop built into it which checks the value of a second counter, *ST4*. *ST4* keeps track of the number of APGF recursions that have been computed. After the first three recursions, control returns to state 17 where another recursion of the same filter is computed. Upon completion of the fourth recursion, the multipliers are initialized for a new filter and a jump made to state 16 to begin the execution of the new filter.

Finally, when all of the filters have processed the new data point, the signal *RX* is asserted indicating that the RWM address has advanced beyond the physical memory. The assertion of *RX* is detected by the state machine which then returns to state zero. Careful observation of the state table reveals that the multipliers were loaded and multiply operations initiated prior to the completion check. The load-before-branch allowed for the elimination of at least one state from inside the filter loop. Since the memory address pointer has been advanced beyond the bounds of memory, no valid data is loaded into the multipliers. Therefore upon returning to state zero the data is simply discarded with no impact on system performance.

The state machine diagram is shown in Figure 33. Inside the state machine is a five-bit latch which serves as the state counter. The state counter latch is conditionally loaded with either the next sequential state or a jump state. The next sequential state is computed by incrementing the present state. The jump address is calculated based on the present state and the *RX* signal. To aid in understanding the jump calculation, Table 9 lists the three cases where jumps occur.

Table 9: Filter State Jump Addresses

Present State	Representation	Next State	Representation	Condition
21	10101	17	10001	$MP8 \cdot ST4$
25	11001	16	10000	<i>RX</i>
25	11001	0	00000	<i>RX</i>

It can be observed from Table 9 that the center three bits are always '0' after a jump. Likewise note that the least significant jump bit can be determined by simply using the center bit before the

jump. Finally, because  $RX$  remains '0' until all filters have finished processing the current input value, its inverse ( $\overline{RX}$ ) may be used as the most significant bit of the jump address.

In order to fully reset the filterbank, all RWM cells must be cleared. To accomplish the reset function, a reset state counter has been incorporated into the design of the finite state machine. Upon receiving an asserted reset signal, the reset counter asserts an internal reset to the rowselect circuits which will last for two clock periods. If the externally applied reset signal is held asserted, and the clock is cycled, the internal reset signal will be removed and the rowselect circuitry will begin to cycle through the RWM rows at a rate of one per clock cycle. At the same time, the write amplifiers of the RWM will drive '0's into the selected RWM cell. To accomplish a full system reset, the externally applied reset signal must remain asserted while the clock is cycled at least one time for each row in the memory array.

The final design issue of the filterbank design is its output handshaking signals. Like the input to the filterbank, the output is also asynchronous. Of the three functional divisions of the overall architecture, the filterbank stage requires the most time to execute. Consequently, the stages that follow will enter a low-power idle state while waiting for data by inhibiting the clocking of all internal signals.

When the filterbank completes any given filter's computation, it latches the data into its output register denoted as "compress latch" in Figure 32. After the data has been latched, the filterbank's controlling state machine asserts  $DXRQ$  (data transmit request) to request to send data to the compression stage which follows. Upon receiving the asserted  $DXRQ$  signal, the compression unit responds by asserting  $DXAK$  (data transmit acknowledge) which causes the release of  $DXRQ$ . A discussion of the compression unit follows in Section 4.3.

Figure 34 illustrates the output of the VHDL implementation of the filterbank for 12 channels with center frequencies ranging from 200 Hz to 7 kHz. The envelope of both the middle-ear IIR filter (as seen in the peaks of the responses) as well as each of the APGF filters are clearly evident in Figure 34. The companion plot generated by AIM using the default options is shown in Figure 35. The difference in magnitude in the filters near 2000 and 3000 Hz is attributed to the removal of the middle-ear resonance from the approximation model.

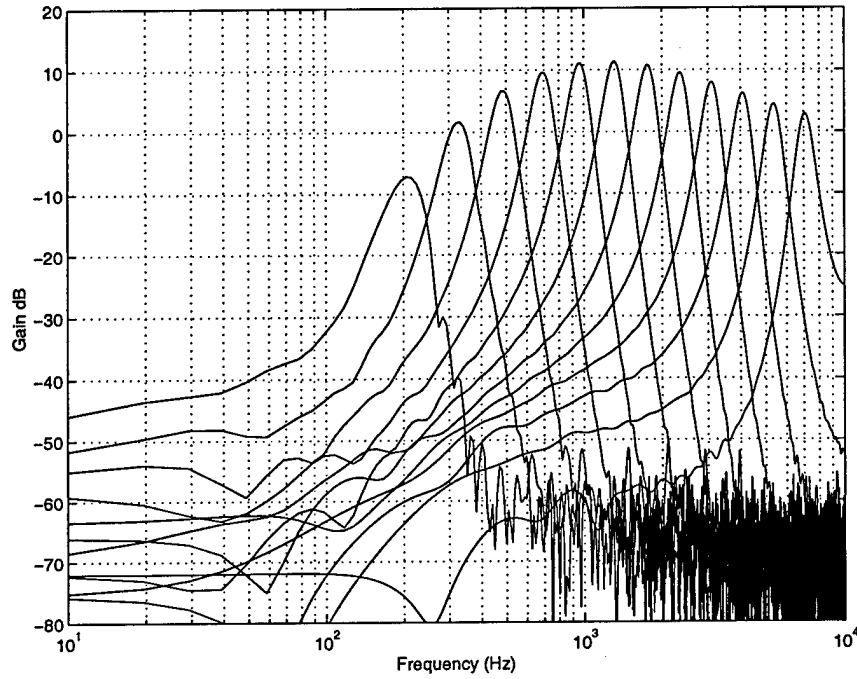


Figure 34: Frequency Response of 12 Channel VHDL Filterbank

### 4.3 Amplitude Compression

Following the filtering of the data in the filterbank, the data is passed on to the amplitude compression stage which accomplishes two tasks: a half-wave rectification, and conversion of all input values to millibels. Unlike the filterbank, the compression stage logic is fully combinational and does not require state machine control. By definition, the millibels of any value  $X$  is:

$$X_{\text{millibels}} = 2000 \times \log_{10}(X) \quad (24)$$

As discussed in the previous chapter, the  $\log_{10}(X)$  can be re-written as (rounding  $\log_{10}(2)$ ):

$$\log_{10}(X) = 0.30103 \times \log_2(X) \quad (25)$$

and an approximation for  $\log_2(X)$  has already been established. Therefore, combining these two expressions, the circuit must be able to compute:

$$X_{\text{millibels}} = 602.06 \times \log_2(X). \quad (26)$$

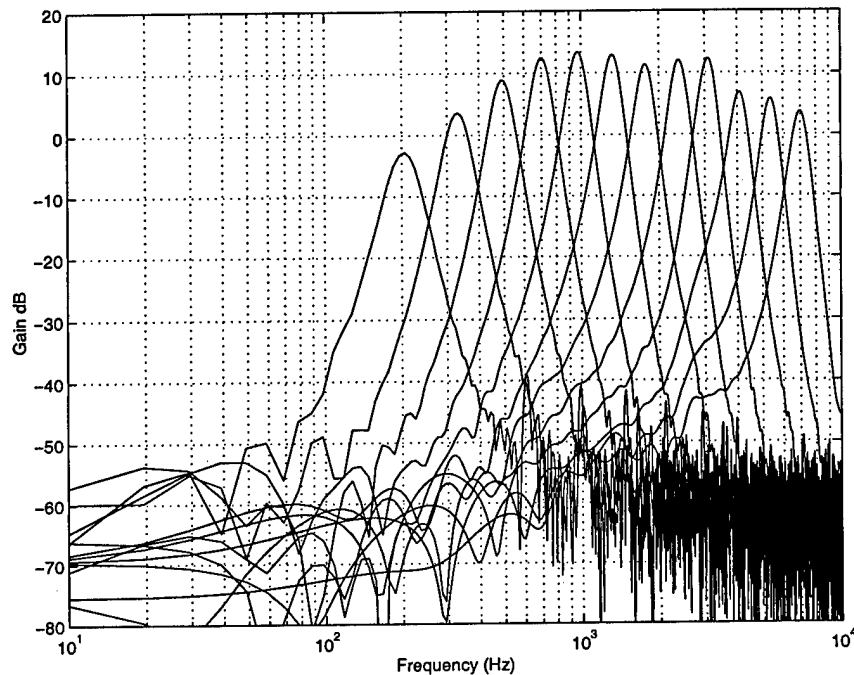


Figure 35: Frequency Response of 12 Channel AIM Filterbank

The approach implemented was to first compute the logarithm in base two using the approximated technique discussed in Chapter 3 using a bit shifter. The approximate logarithm is then adjusted for error reduction through an arrangement of adders. Finally, the result of the logarithmic approximation is multiplied by 602 through three additions. The following three subsections provide the details of implementation for each of these operations.

**4.3.1 Bit Shifter.** The first stage of the compression unit is the bit shifter which very quickly computes Mitchell's approximation of the  $\log_2$  using decoders, a barrel shifter, and a small amount of ROM. The decoders determine the location of the most significant '1' bit and passes this information on to the characteristic ROM and the barrel shifter. The barrel shifter and the ROM work in parallel to produce Mitchell's approximation to the  $\log_2(X)$ . The discussion of these three components of the first stage in the conversion to millibels follows.

**4.3.1.1 Decoders.** Central to the  $\log_2$  approximation is the determination of the bit position of most significant logical '1'. Traditional methods for the location of the most significant '1' involved shifting and counting[28, 29]. Here, special purpose decoder cells were designed

to determine the position combinationally. The decoders are similar in function to the generate-propagate circuitry of look-ahead adders which speed-up the carry path. In this design, however, the circuit is used to determine the location of the most significant non-zero value.

Figure 36 shows the basic logic of the new decoder cell. The  $GENERATE_n$  signal is propagated downward from the most significant to the least significant bit position. The  $GENERATE$  input of the most significant bit is fixed at a logical '1'. At any general stage, a  $GENERATE_n$  input of '1' indicates that there is not a cell of higher significance with a logical '1' on its data input. The presence of a '1' on  $GENERATE_n$  enables the present cell to check its data input ( $D_n$ ).

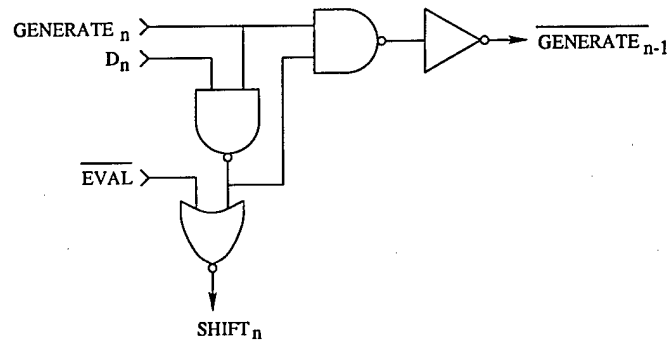


Figure 36: Bit Position Decode Logic

If  $D_n$  is a '1', and  $GENERATE_n$  is a '1', then the  $SHIFT_n$  output becomes a '1' and the  $GENERATE_{n-1}$  output is driven low. The  $SHIFT_n$  output indicates that the generating cell is the most significant bit that contains a '1' input. The driving of  $GENERATE_{n-1}$  to '0' inhibits all lower cells from asserting their  $SHIFT$  output.

The signal  $\overline{EVAL}$  is incorporated to ensure that there are no simultaneous firings of the  $SHIFT$  signals. To understand the need for the  $\overline{EVAL}$  signal, consider a case where the input contained all '0's. In this case, a  $GENERATE$  signal of '1' will propagate through all of the cells and none of the cells will fire. If the next value sent to the decoder contains multiple '1's, every cell which received a  $D_n$  of '1' will momentarily assert its  $SHIFT$  output. Shortly thereafter, the higher cells will inhibit all lower cells and only the most significant  $SHIFT$  signal will remain asserted.



The initial instability of the *SHIFT* signals in the case just discussed is unacceptable due to the domino effect it has on the circuits which follow. The addition of the  $\overline{EVAL}$  adds needed stability to the generation of the *SHIFT* signals. While  $\overline{EVAL}$  is high, no *SHIFT* outputs are produced but the cells are able to decode the input. When enough time has elapsed to ensure the stability of the decoders,  $\overline{EVAL}$  is dropped to zero and the appropriate *SHIFT* signal is generated.

A second look at the cell in Figure 36 reveals two gate delays in the *GENERATE* signal path. One gate delay can be removed by removing the inverter from the cell and creating a complementary cell as diagramed in Figure 37. The inverter that was removed from the first cell is incorporated into the second cell but is no longer in the *GENERATE* path. To correctly handle the now inverted *GENERATE*, the NAND gate is replaced with a NOR gate in the modified cell. The multi-bit decoder is implemented by alternating the two cells beginning with the cell of Figure 36 in the most significant bit position.

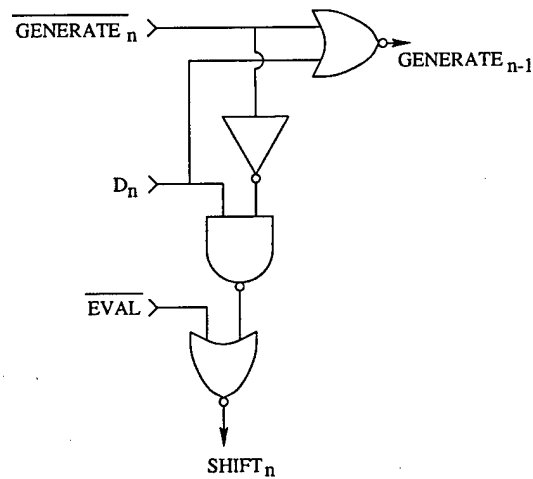


Figure 37: Complementary Bit Position Decode Logic

There are two special cases to address in the calculation of the logarithm. The first case is that of a zero input value. In this case none of the decoder cells will fire leaving a logical '1' on the  $GENERATE_0$  signal. When  $GENERATE_0$  is asserted an error signal is produced which in turn forces all output bits to '0'.

The second special case occurs when a negative input value arrives. As in the case of a zero, the logarithm of negative numbers is not defined. Since the input is a two's complement binary

number, the identification of negative numbers is accomplished by checking the most significant bit on the input. If the most significant bit is set, all bits fed into the decoder are forced to zero. The all-zero input prevents any of the *SHIFT* signals from firing. One of the two subsystems which use the *SHIFT* signals is the barrel shifter, which is discussed next. The second consumer of the decoder output is the ROM, which is discussed in Section 4.3.1.3

**4.3.1.2 Barrel Shifter.** The products of the decode stage are the *SHIFT* signals. If the input number has a logarithmic representation, only one of these *SHIFT* signals will be active. The active signal will cause the input value to be rotated to the left until the most significant '1' is positioned in the left-most bit position. A barrel shifter is used to perform the bit rotation.

As seen in Figure 38, the barrel shifter is comprised of only N-channel transistors arranged in a two-dimensional array. For the purpose of illustration, an array of only 3 bits is shown. The input data enters the array on vertical data lines and is passed to the horizontal output data lines by the N-channel transistors. The *SHIFT* signals from the decoder determine the routing through the N-channel transistors. In all cases, the data passes through only one transistor in order to be transferred from an input to an output line. Since all of the transistors in the array are purely passive, the only power consumed by the shift operation is due to the capacitive load the array places on its input.

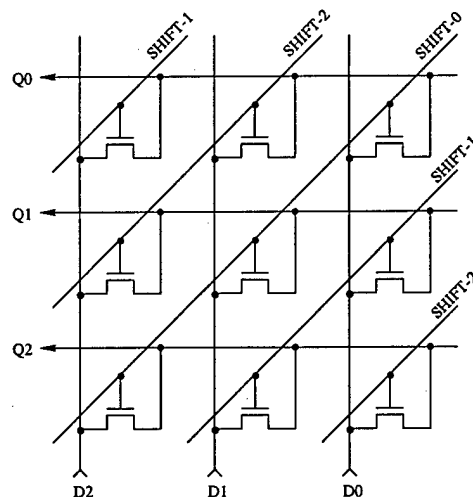


Figure 38: Barrel Shifter Array

After the data passes through the barrel shifter, it is aligned so that the position of the most significant '1' is always on the same output line. The realignment is necessary so that the remaining bits may be appended to the integer portion of the logarithm (the characteristic) which is read from a small ROM bank. The next section provides a discussion of the generation of the characteristic using a ROM bank.

**4.3.1.3 Characteristic ROM.** In Mitchell's approximation the characteristic of the logarithm is determined by counting bits to find the most significant '1'. In contrast, using the decoders discussed Section 4.3.1.1, a unique *SHIFT* signal is generated for each possible location of the most significant '1'. The *SHIFT* signals could be processed through an encoder to determine the characteristic, but a circuit that consumes much less power and space is a read-only memory (ROM). Since only one of the decoder cells will generate a *SHIFT* output at any time, the active *SHIFT* output may be used to access rows within a ROM that is loaded with the logarithm characteristics. Optionally, the characteristic could be generated from combinational logic based on the *SHIFT* signals. A ROM was chosen in this case to eliminate the need for P-channel transistors, thus allowing for a physically smaller structure.

The amount of ROM required for the logarithmic approximation is minimal. For two's complement input data of width  $N$  bits, the required ROM bank is  $\log_2(N) \times N$  bits. There must be one row of ROM for each input bit, and each row must contain  $\log_2(N)$  bits to store the characteristic for that bit position.

The last row in the ROM (corresponding to bit  $N - 1$ ) is actually associated with the sign bit. By loading the last row in the ROM with '0's and forcing the input bits to '0' when a negative number is being processed (as previously discussed), the logarithm generated for any negative number will be zero. Through defining the mathematically undefined logarithm of negative numbers as zero, the circuit performs the half-wave rectification required by the neural encoding algorithm. A logarithm error signal is generated by the hardware for zero and negative inputs, but is ignored in this application to allow the half-wave rectification to proceed.

After the input value has been shifted, and a characteristic is chosen from the ROM, the two are combined to form Mitchell's approximation. The result is 4 bits of unsigned integer, and

12 bits of fixed-point binary fraction which is sent to the adjustment circuit where the second approximation to the logarithm is applied.

*4.3.2 Logarithm Adjuster.* Following the half-wave rectification and the implementation of Mitchell's approximate logarithm, the data is adjusted to improve the accuracy of the logarithmic approximation. The adjustment to Mitchell's approximation is accomplished by adding a shifted copy of a portion of the fraction to the original approximation. The optimal number of bits to shift is four, and the optimal distance to shift them before the addition is 2 bit positions to the right. (See Section 3.4.2 for derivation.) If the most significant bit of the fraction is '0', the shifted bits are added as they are. However, if the most significant bit of the fraction is a '1', the shifted bits are inverted prior to the addition.

In the VLSI implementation, the shift is performed through wire routing. The conditional inversion is carried out by three XOR gates that combine the most significant fraction bit with the next three bits respectively. If the most significant bit is set, the others become inverted. It is unnecessary to conditionally invert the most significant bit, because the XOR of any bit with itself is always '0'.

After the conditional inversion of the bits, they are added back to the original approximation. The addition requires three full adders (FA), and three half adders (HA) as shown in Figure 39. The right-most half adder accounts for the most significant adjusted bit, which is always zero. The second and third half adders, feeding  $Q_{10}$  and  $Q_{11}$ , account for the two most significant bits in the original fraction which only need to be adjusted by the carry from the lower bits. The carry must not propagate beyond the most significant bit of the fraction. If permitted, the characteristic would be altered, introducing large errors.

*4.3.3 Conversion to Millibels.* The output from the adders is the approximate base-two logarithm of the input value. Additionally, the data stream is half-wave rectified through the way in which the circuits deal with negative values as previously discussed. The next step is to convert the base-two logarithm into millibels.

As indicated in Equation 26, the conversion to millibels requires multiplying the base-two logarithm by 602.06, which is rounded to 602 so that it may be handled as an integer. In binary

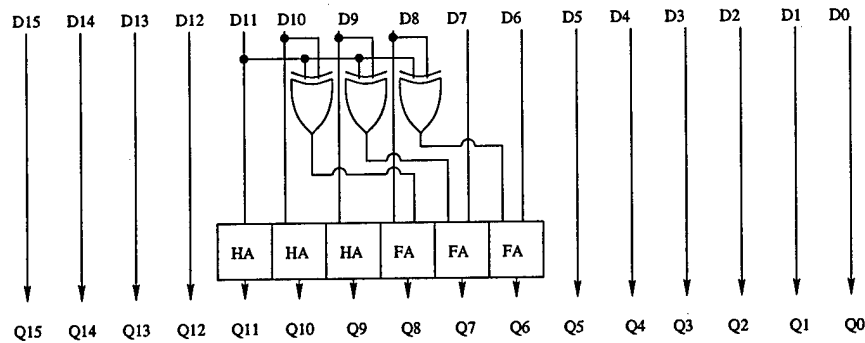


Figure 39: Logarithm Adjusting Circuit

form, 602 is represented by the bit sequence  $1001011010_2$ . Two approaches to implementing the multiplication were considered. The first approach utilizes a multiplier and a small state machine to control its operation. An implementation using only combinational logic (adders) was also investigated. After comparing the size, speed, and power requirements, it was obvious that the adder-only solution was the more favorable option.

The adder-only solution for the conversion to millibels has an advantage over a multiplier because there is a repeated pattern in the binary representation of 602 which can be used to simplify the computation. Note the pattern  $101_2$  which appears twice in the binary representation of 602. Since any multiplication can be accomplished by adding and shifting, we can accomplish the partial product for the pattern  $101_2$ , then shift the sum to form the second partial product without an additional adder.

To illustrate, let the base-two logarithm of an arbitrary number be represented by the sequence  $ccccffffffffff$  where the  $c$ 's represent the characteristic integer, and the  $f$ 's represent the fractional part of the approximate log. Expanding the multiplication of the arbitrary value by 602 we get (neglecting the '0' bits in 602):

$$\begin{array}{r}
 ccccccccccccccc \\
 \times 0000001001011010 \\
 \hline
 ccccccccccccccc0 \\
 ccccccccccccccc000 \\
 ccccccccccccccc0000 \\
 ccccccccccccccc00000 \\
 ccccccccccccccc000000 \\
 + ccccccccccccccc00000000 \\
 \hline
 iiiiiiiiiiiiiiiiffffc
 \end{array} \tag{27}$$

In Equation 27 note that the first two partial products are the same as the second pair, with the exception that the second pair is shifted to the left by three bits. Therefore, the sum of the first two terms can be added to a shifted version of itself, achieving in two addition operations the equivalent of three. The second sum is then added to the final term to yield the final answer. The adder arrangement for these operations is illustrated in Figure 40.

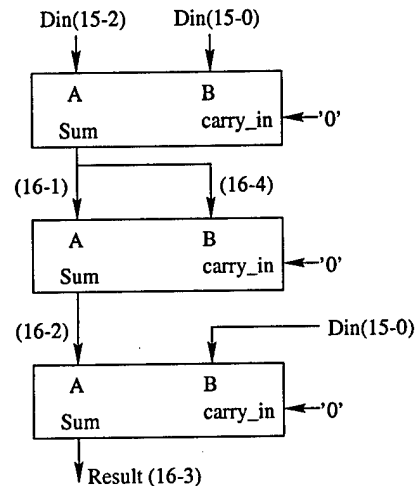


Figure 40: Optimized Adder Tree for Millibel Multiplication

A second optimization can be made to reduce the size of the adders required. The final result that will be passed to the next stage of the architecture will be an integer value, represented by the  $i$ 's in Equation 27. Therefore we only need to maintain enough fractional bits to provide adequate rounding precision for the addition operations being performed.

Note that the three least significant bits of the first two terms may be omitted from the addition because these bits in the second term '0's, and will therefore not cause a carry to the fourth bit. For the same reason, when adding the sum to a shifted form of itself, the three least significant bits may again be dropped. Finally, in the last addition there are three zero terms which likewise can be used to reduce the bits in that addition.

The first addition requires a sixteen bit adder in order to preserve the numeric integrity. Following the first adder, each successive stage requires one additional bit since the carry output of each preceding stage becomes an input bit of the next stage. However, because the end result is to be an integer, we can make the system symmetric using only 16-bit adders by truncating the least

significant fraction bit from each sum after the first adder. In Figure 40 the numbers in parentheses represent the bit positions being propagated.

The output of the final adder in Figure 40 is fourteen bits. Because of the multiplication being performed, the upper two bits will always be '0', and are not computed. To re-establish the result as a sixteen-bit value consistent with the rest of the hardware, two '0' bits need to be concatenated onto the most significant end of the final sum.

The last phase of the compression stage is the generation of the asynchronous hand-shaking signals for the adaptive thresholding stage which follows. Data arrives to the compression stage at a rate of one value every 48 clock cycles when the filterbank is processing.

Since the compression stage is fully combinational, its computation could be completed in a single machine cycle. However, if implemented in this manner the overall system clock would need to be slowed to allow for the completion of the compression computation. Therefore, a counter arrangement was designed that guarantees enough time between the arrival of the data from the filterbank and the generation of the send request for the adaptive thresholding stage. Figure 41 illustrates the design.

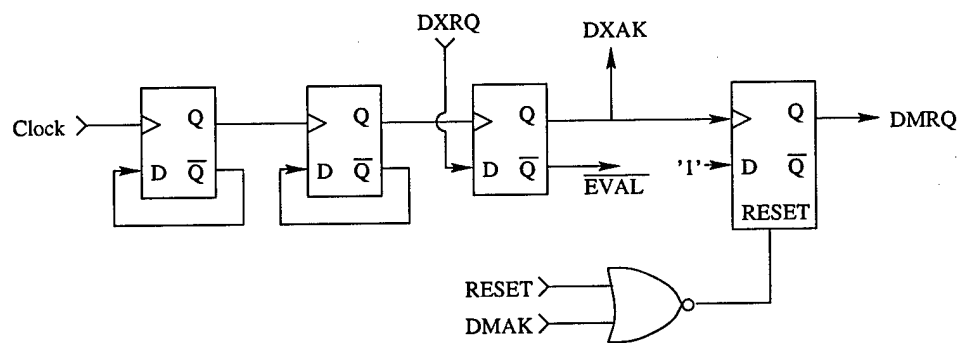


Figure 41: Handshaking for Compression Stage

In Figure 41, *Clock* represents the shared system clock. *DXRQ* and *DXAK* are the required handshaking signals for the filterbank interface, while *DMRQ* and *DMAK* are the handshaking signals to and from the adaptive thresholding stage. Essentially, the circuit of Figure 41 divides the master clock by four. When data arrives from the filterbank, the next falling edge of the divided clock asserts the acknowledge (*DXAK*) signal for the filterbank. *DXAK* remains asserted for four master clock cycles, then resets.

The falling edge transition of *DXAK* loads the final latch which sends *DMRQ* to the adaptive thresholding stage. The thresholding stage responds by asserting *DMAK*, placing the compression stage back into its idle state. Finally, to ensure data isolation between input and output, a 16-bit data latch is included at the output. The output latch is clocked by the falling edge of *DXAK*, and is loaded at the same time that *DMRQ* is generated.

An alternative solution to the clocking problem just discussed would be to absorb the compression stage into the filterbank. This is possible because the time required by the compression is small compared to the filtering. The time to compress the data stream from one filter could be overlapped with the beginning of the processing of the following filter. Therefore, no additional states would be required to complete any filter. The one drawback to this approach is that in order to compress the data from the last filter in the filter bank, additional states would need to be added into the overall state machine, thus increasing its complexity slightly.

The three sections of the compression stage, working under the control of asynchronous handshaking signals for data input and output, convert the output of the filterbank into a compressed form. Specifically, on exiting the compression stage the data has been half-wave rectified and logarithmically compressed into millibels. The compression is actually the first phase of the neural encoding process which accounts for the mechanics of the inner hair cell stimulation. The actions of the inner hair cell, how they convert the stimulation into electrical nerve events, is discussed in Section 4.4.

#### 4.4 Neural Encoding

The filterbank models the conversion of the sound energy from the acoustical realm into the motion of the basilar membrane. The compression stage which follows further models the stimulating of the inner hair (nerve) cells. The final stage of the auditory system models the behavior of the inner hair cells which convert the mechanical energy into electrical pulses, a process known as neural encoding.

The third stage of the hardware design combines the adaptive thresholding algorithm and an integration filter to form the neural encoding processor (NEP). The combination of functions allows for maximum utilization of the computational units. The combination of algorithms is



possible because the two operations can be performed in less time than is required for each stage of the filterbank.

The approach used to design the NEP was similar to the approach used in designing the filterbank. The algorithms outlined in Chapter 3 were sequentially analyzed, then the required components were selected. The components were interconnected using multiplexers and latches as necessary. Finally, a state table was derived indicating how each of the multiplexers, latches, and functional units were to be controlled. The state table was processed by Espresso<sup>3</sup> to generate optimized state equations. Verification of the design was performed using a structural VHDL description.

All of the individual components of the NEP were laid out in a custom VLSI implementation. Figure 42 presents the final design of the NEP showing how the adaptive thresholding and integration filtering are combined in the proposed implementation. As discussed in Section 3.5, the adaptive thresholding process models the way in which the inner hair cells require a recovery period after firing. Since the model used for generating nerve firing events has an undesirable rapid return-to-zero, the integration filter is included to shape the output pulses. While both the adaptive thresholding and integration filtering are functions of the NEP, the discussion which follows separates the two for clarity. The first to be discussed is the implementation of the adaptive thresholding.

*4.4.1 Adaptive Thresholding.* The approximate adaptive thresholding algorithm, developed in Section 3.5, is detailed by the code shown in Figure 24. Recall the purpose of the adaptive thresholding is to only allow input events which exceed the current firing threshold to cause an output event. Thresholding, therefore, models the recovery period which the inner hair cells enter after firing.

To begin the discussion of the hardware implementation of the approximated adaptive thresholding algorithm, consider the the current threshold variable. One read-write memory storage is required for each channel of the filterbank in order to hold that channel's current threshold. When new data arrives, the current threshold is unconditionally decayed.

---

<sup>3</sup>Espresso is one of the tools in the Berkeley VLSI tools distribution. It accepts a state table as input and generates optimized and reduced state equations and PLA programming tables as output.



In the approximate algorithm the threshold (*microphonic* in Figure 24) is adjusted by the expression:

$$\text{microphonic} = \text{microphonic} \times (1 - \text{rapid\_decay}) + (\text{absolute\_limit} \times \text{rapid\_decay}). \quad (28)$$

As previously discussed, two simplifying approximations were applied to equation 28. First, *absolute\_limit* is approximated by the constant 1024. Second, *rapid\_decay* requires 12 bits of storage and is always a positive fraction less than one. Therefore, the quantity  $(1 - \text{rapid\_decay})$  is simply the two's complement of *rapid\_decay*.

Considering these facts, a simplification to the hardware may be made by storing the two's complement of *rapid\_decay*. The first way to simplify the hardware is to reduce the ROM for the storage of *rapid\_decay* to 11 bits since in the two's complement representation the leading bits will always be '1', indicating it is a negative value.

Secondly, the multiplication by *absolute\_limit* is the same as a shift to the left of 10 bit positions of the two's complement of the stored value for *rapid\_decay*. However, after multiplying by *absolute\_limit*, the only significant bits are those which are to the left of the fixed decimal point. Therefore, the two's complement can be approximated by inverting *rapid\_decay*. The inversion is shown between the *rapid\_decay* ROM and MUX8 in Figure 42.

In the design, there is one memory address counter which is shared by three banks of RWM and two banks of ROM. All data required for the processing of one channel is stored at the same memory address in each of these banks. Therefore, the threshold decay can be computed by fetching, then multiplying the current threshold (fetched from RWM bank 2) by the two's complement of *rapid\_decay* (fetched from the second ROM bank). The multiplication is performed using an array of 18 of the multiplier slice cells used in the filtering stage.

The result of the multiplication is then added to the complement of the stored *rapid\_decay* that has been shifted by 10 bits in order to approximate the multiplication by *absolute\_limit*. The resulting sum is the new decayed value for the threshold. Since the computed value is needed again and may be changed in the algorithm, it is placed into a temporary latch (MIC in Figure 42) instead of being written back to RWM. Notice that all logic cells pictured below the multiplier in Figure 42 include one additional fraction bit for the purpose of rounding.

After the threshold is processed for decay, it is compared to the new input value to see if the input is strong enough to cause an output event. The comparison is accomplished by cycling the decayed threshold from MIC into the arithmetic logic unit (ALU) where it is subtracted from the input value. The difference is held in the register labeled TMP for later use. If the result of the subtraction is negative, no output event occurs because the input did not exceed the threshold. The negative result is detected by observing the most significant bit of the result. When the result is negative, the decayed threshold is written back to RWM, and a zero is passed to the integration filter step.

Conversely, when the input exceeds the threshold, the result of the subtraction just mentioned is positive and an output event occurs. Upon detecting a positive subtraction result, then the approximate algorithm first averages the current threshold and the input. The average becomes the new threshold which is written back to RWM. Secondly, the difference held in the TMP latch is multiplied by the compensation factor for the channel. The compensation constants are stored as 8-bit values which represent 6-bits of unsigned integer and 2-bits of fraction. The product of the compensation is then stored in TMP and only the integer portion is passed to the integration filter. The additional fraction bits are only used by the ALU for rounding.

**4.4.2 Integration Filter.** An integration filter completes the design of the neural encoding processor. The integration filter is a second-order IIR filter implemented as a recursive first-order IIR filter. Each channel is filtered by an identical filter, thus only one set of ROM constants are needed. However, state information from the previous two outputs must be maintained for each filter channel. The storage of state information is indicated in Figure 42 by the RWM banks labeled RAM0 and RAM1. The addresses of these memory locations are aligned with those of the corresponding thresholding values so that an address change is not required to filter the thresholding outputs.

The two constants for the integration filter, common to all filter channels, were hard-coded on the inputs to a 4-to-1 multiplexer. These values are labeled L and K in Figure 42. As derived in Chapter 3, both L and K can be approximated by 4-bit fractions. While 4-bit integers can be multiplied using Booth's algorithm in two cycles, fractions such as these require three cycles so that a leading '1' in the fraction is not incorrectly interpreted as a negative sign bit.

The IIR filter operates by multiplying the previous output (fetched from RAM0) by the constant L. The result is then temporarily stored in the MIC register while the input to the filter is multiplied by the constant K. The input to the filter is the value left in TMP by the thresholding algorithm. When the second multiply is completed, the result is added to the first product and the sum is stored in TMP and written back to RAM0. The algorithm then repeats using RAM1 rather than RAM0. The TMP register holds the final output after the completion of the second cycle of the algorithm.

In order to sequence both the adaptive thresholding and integration filter algorithms through the NEP, a control circuit is required. The discussion of the circuit which controls these two algorithms follows in the next subsection.

**4.4.3 Control.** As with the filterbank, the control for the NEP is accomplished by a finite state machine. Twenty-one states were necessary to complete the processing required by the NEP. The state machine, using five state bits, incorporates one conditional jump, four multiply loops, and one control loop. The state assignments are sequential with the exception of the last state (22) which has been chosen to reduce the jump logic.

Figure 43 is a block diagram of the finite state machine for the NEP. The states which are sequenced by the machine of Figure 43 are tabulated in Table 10. State 0 is the idle state and is

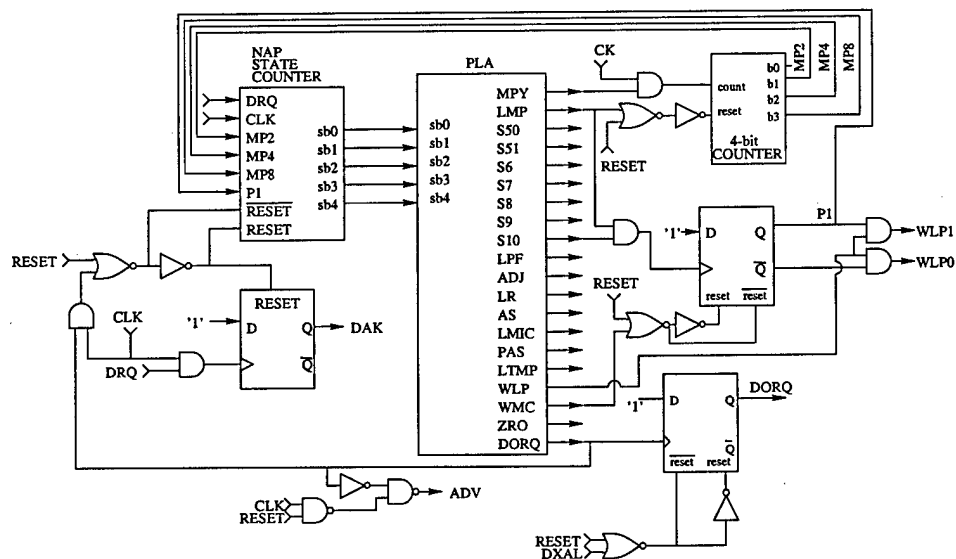


Figure 43: State Machine for the Neural Encoding Processor

maintained until the signal DRQ is asserted by the compression stage. When DRQ is asserted, the NEP begins by computing the threshold decay in states 1 through 5. The input is subtracted from the current threshold in state 6 and the sign of the result is checked. If the result is negative, a jump to state 22 is taken which clears the output register and writes the decayed threshold back to RWM. State 22 then jumps to state 13 where the integration filtering begins. The equations which define the output bits of the state table are listed in Appendix C.

Table 10: Neural Encoding Processor State Table

State	lin	lmp	s50	s51	s6	s7	s8	s9	s10	mp2	lpf	adj	lr	as	lmic	pas	ltmp	wlp	wmc	zro	dorq	Next
0	0	0	.	.	.	.	.	.	.	0	.	.	0	.	0	.	0	0	0	0	0	1
1	1	0	.	.	.	.	.	.	.	0	.	.	0	.	0	.	0	0	0	0	0	2
2	0	1	0	1	1	.	.	.	.	0	0	.	0	.	0	.	0	0	0	0	0	3
3	0	0	.	.	.	.	.	.	.	1	.	.	0	.	0	.	0	0	0	0	0	4:mp6
4	0	0	.	.	.	.	.	.	.	0	.	0	1	.	0	.	0	0	0	0	0	5
5	0	0	.	.	.	0	0	.	.	0	.	.	0	0	1	0	0	0	0	0	0	6
6	0	0	.	.	.	1	1	.	.	0	.	.	0	1	0	0	1	0	0	0	0	7:pos 7:neg
7	0	1	0	0	0	.	.	1	.	0	0	.	0	.	0	.	0	0	0	0	0	8
8	0	0	.	.	.	.	.	.	.	1	.	.	0	.	0	.	0	0	0	0	0	9:mp4
9	0	0	.	.	.	.	.	.	.	0	.	1	1	.	0	.	0	0	0	0	0	10
10	0	0	.	.	.	0	.	.	.	0	.	.	0	0	0	1	1	0	0	0	0	11
11	0	0	.	.	.	1	1	.	.	0	.	.	0	0	1	0	0	0	0	0	0	12
12	0	1	1	0	1	.	.	0	0	0	1	1	0	.	0	.	0	0	1	0	0	13
13	0	0	.	.	.	.	.	.	.	1	.	.	0	.	0	.	0	0	0	0	0	14:mp2
14	0	0	.	.	.	.	.	.	.	0	.	0	1	.	0	.	0	0	0	0	0	15
15	0	1	0	0	0	0	0	1	0	0	1	.	0	0	1	1	0	0	0	0	0	16
16	0	0	.	.	.	.	.	.	.	1	.	.	0	.	0	.	0	0	0	0	0	17
17	0	0	.	.	.	.	.	.	.	0	.	0	1	.	0	.	0	0	0	0	0	18
18	0	0	.	.	.	0	1	.	.	0	.	.	0	0	0	0	1	0	0	0	0	19
19	0	1	1	0	1	.	.	1	1	0	1	.	0	.	0	.	0	1	0	0	0	13:p1 20:p1
20	0	.	.	.	.	.	.	.	.	0	.	.	0	.	0	.	0	0	0	0	1	0
21	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
22	0	1	1	0	1	.	.	0	0	0	1	0	0	.	0	.	0	0	1	1	0	13

If the result of the subtraction in state 6 is positive, then the difference is multiplied by the compensation factor in states 7 through 9. Note that because the fraction of the compensation factor is not aligned with that of the other constants, an adjustment (shift) is required on the output of the multiplier to re-align the bits. The result is passed through the ALU and stored in TMP as the next input to the integration filter (state 10).

The processing continues by raising the current threshold in states 11 and 12. Finally, states 13 through 20 compute one pass of the recursive integration filter. The state machine output *P1* is set after the first pass through the integration filter when a jump back to state 13 is made. On the second pass, the asserted *P1* indicates that the filter is in its second pass and thus, upon reaching state 20, the output request DORQ is asserted and the machine returns to the idle state.

All of the program jumps are accomplished using a jump loader. In order to determine the most efficient state numbering, a study of the current state and jump states was performed. As a

start, since the binary representations of 6 and 22 differ by only one bit, the jump logic is simplified by choosing 22 as the destination of the jump in state 6. Table 11 tabulates the current and next states for all of the jumps required in the NEP.

Table 11: Neural Encoding Processor Jump States

Current State	Next State	Jump Vector
6 00110	22 10110	$\overline{sb4} \ sb4 \ 1 \ sb4 \ sb4$
19 10011	13 01101	$\overline{sb4} \ sb4 \ 1 \ sb4 \ sb4$
22 10110	13 01101	$\overline{sb4} \ sb4 \ 1 \ sb4 \ sb4$

In each case shown in Table 11, the target of the jump can be computed as a function of only the most significant state bit,  $sb4$ . The jump vector is hard-wired into the multiplexer of the state counter pictured in Figure 44. The state counter consists of a 5-bit data latch which is conditionally loaded with one of the following: the next sequential value, the jump-load vector (LV), or left at its current value. The signal STEP determines whether or not the latch is reloaded, and the signal LD determines whether either the next sequential state or the jump vector is loaded.

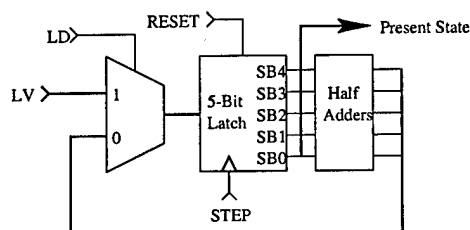


Figure 44: Neural Encoding Processor State Counter

The logic to generate the LD and STEP signals, as well as all of the control signals in the state machine pictured in Figure 42 are generated by a programmable logic array (PLA). Optimized equations for the PLA were generated by processing the state information of Table 10 through Espresso. In the VHDL model, these equations were implemented in behavioral form assuming a NAND-NAND PLA structure.

#### 4.5 Summary

Chapter IV provided an overview of the hardware implementation. The issues addressed were those of particular interest due to their unique nature or implementation. Because of the size

of the overall architecture, and because the implementation is fully detailed in the VHDL structural description, the discussion was intentionally incomplete. Appendix E provides the top-level structural description of the proposed architecture corresponding to the system block diagrams given.

As documented, the modifications to the algorithms developed in Chapter 3 have been mapped to a VLSI implementation. Additionally, using current technology, the proposed design will fit onto a single VLSI chip simplifying its integration into a variety of systems. For example, as discussed in Chapter 5, if implemented in  $0.8\ \mu\text{m}$  CMOS VLSI, the entire circuit would fit onto a 3 mm-square chip.

The proposed architecture can be characterized as a multi-rate design. The data enters the processor at a sample rate determined by the system to which it is attached. Internally, the data is then processed at a much higher clock frequency where it is split into multiple frequency channels. As a result, the output is time-division multiplexed and exits the system at a rate higher than the input data rate. The output data is time-division multiplexed on a processing channel basis. Therefore, the time between any two consecutive samples belonging to the same channel will be equal to the period of the input data.

The designed system is also asynchronous in its interface to other devices. All data entering or leaving the architecture is controlled by hand-shaking signals versus a clock. Internally, however, all processing is sequenced by a clock operating much faster than the data rate.



## *V. Testing and Evaluation*

### *5.1 Introduction*

Having completed the design of a processor to execute the approximate AIM algorithms, the concluding task was to validate the proposed architecture. This chapter presents the results of the testing and evaluation accomplished during the validation phase. The analysis of the proposed system is divided into four sections: a functional comparison of the proposed architecture to AIM, and estimations of the architecture's speed, size, and the electrical power required for operation.

### *5.2 Functional Comparison*

Perhaps the most important measure of the proposed architecture (and its underlying algorithms), was a comparison of its performance to that of AIM in phoneme recognition experiments. The VHDL simulation of the hardware architecture required nearly two hours to generate an impulse response of only tens of milliseconds in length. Since 700 NAPs were required for the phoneme recognition experiments, and the length of each was one to several seconds, the use of the VHDL to generate the NAPs was not practical as it would require over 200,000 hours of simulation time. Therefore, modified AIM code was used to conduct the phoneme recognition experiments.

Before the phoneme recognition experiments could be run with confidence, it was necessary to validate the VHDL architecture against the modified AIM code. Validation was accomplished by processing identical data through both the modified AIM code and its VHDL equivalent. After each processing stage, the difference between the outputs was computed and evaluated.

Initially, several discrepancies were identified. One difference, due to an error in the proposed architecture, involved a mis-handled carry flag in the filterbank. The carry error was corrected in the VHDL model. The other inconsistencies were attributed to the use of non-integer arithmetic in the modified algorithms of AIM. In these cases, the modified AIM code was adjusted to execute its algorithms using the same precision as the hardware implementation. Once the adjustments were made, the two models produced identical outputs.

When the outputs of the VHDL and AIM models coincided, the phoneme recognition testing could proceed. These experiments, designed after the works of Patterson et al. and Francis, used

ten spoken sentences from each of ten speakers (100 utterances total)[7, 25]. Seven of the speakers were male and three female.<sup>1</sup>

Testing began by using AIM to generate the neural activity pattern (NAP) for each of the utterances. The minimum frequency channel was set at 350 Hz and the maximum frequency channel at 7000 Hz[25]. The remaining channels, set by AIM's spacing algorithm were: 463.3, 592.3, 739.3, 906.8, 1097.6, 1315.0, 1562.7, 1844.9, 2166.4, 2532.6, 2949.9, 3425.2, 3966.8, 4583.8, 5286.8, and 6087.6 Hz. Using a windowed averaging method, each NAP was divided into overlapping 16 ms windows. Each window was then averaged and labeled to indicate the phoneme which was being spoken during that window. The labeling was accomplished using tag-files which originated from the same database containing the utterances.

The tagged averages from nine of the ten speakers were used to train a Kohonen self-organizing feature map[7]. Once trained, the tenth (untrained) speaker was processed by the feature map to determine how many phonemes could be recognized. Statistics on the map's ability to correctly identify the phonemes were recorded and the training/testing process was repeated using each of the speakers as the unknown voice. In each case the process began by generating a new feature map.

The training/testing procedure was repeated with seven different levels of background noise ranging from an absence of noise to a -21 dB signal-to-noise ratio (SNR). The specified amount of white noise was added to the utterances prior to the generation of the NAPs. The remainder of the technique remained the same. Finally, the experiment was repeated using the approximate form of AIM to generate the NAPs.

Figure 45 illustrates the phoneme recognition rates for both AIM and the modified (approximate) AIM model. Figure 45 clearly shows that while the approximations made to AIM have an impact on the recognition rate, the effect is statistically insignificant<sup>2</sup>. In the worst case the difference between AIM and the approximate model is only 1.2%. The SNRs used in the testing were -21dB, -9dB, -3dB, +3dB, +9dB, +21dB and one case without background noise. Since the

---

<sup>1</sup>All utterances used were from the TIMIT data base. The speakers were mrtk0, mprk0, mwmh0, mjls0, jhpg0, mefg0, mcmj0, fedw0, fcrh0, and fcmm0.

<sup>2</sup>Statistical significance was based on the work of K. Francis using the two-sided Student T distribution with 9 degrees of freedom for a 95% confidence level.[25]

SNR in the case without noise is infinite, the corresponding data point was plotted in Figure 45 as +30dB for convenience.

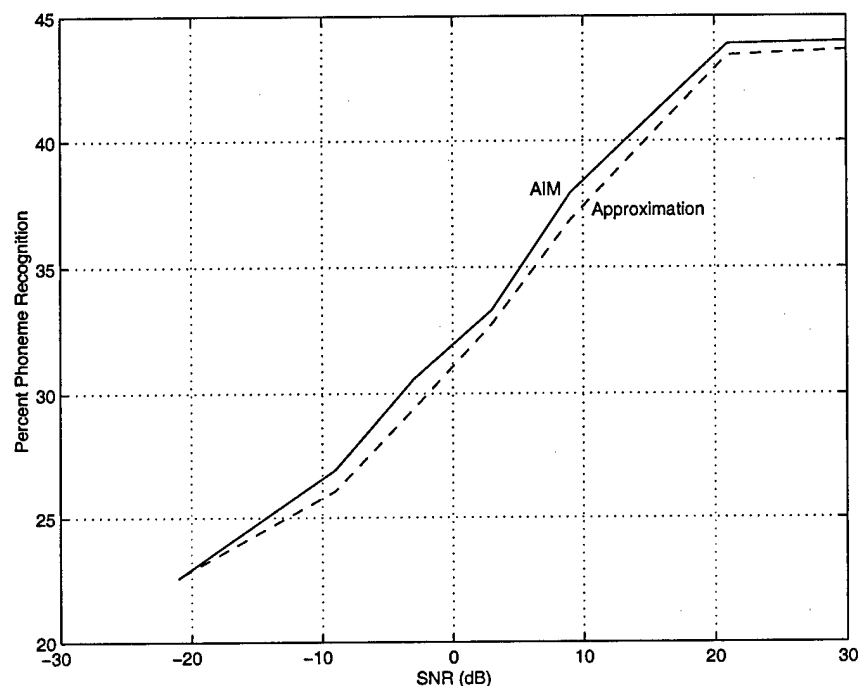


Figure 45: Phoneme Recognition Rates for AIM and Approximation

Although all of the recognition testing was accomplished using the C implementation of the approximation algorithms, the recognition results may be applied equally to the hardware model. The transfer of results is justified because the hardware model was shown previously to be an exact representation of the C code implementation. As stated previously, generating the NAPs directly from the VHDL model was not practical because of the processing time required to model the architecture at the gate level.

### 5.3 Speed

The second criterion used to evaluate the architecture was its processing speed. Because the proposed architecture is primarily a pipelined, sequential machine, data is moved between latches and combinational logic is evaluated between transfers. All logic evaluations must be completed prior to the next clock cycle for the system to function correctly.

As previously discussed, adders are the slowest part of the architecture. Ripple carry adders, used throughout the design for their small size, are inherently slow. The adders found in the filterbank multipliers require more computational time than any other functional unit in the architecture because of their 26-bit length. Even though there are several stages of adders in the logarithmic compression unit, these adders are only 16 bits in length. In addition, the clock in the logarithmic compression unit is one-fourth of the clock in the filterbank to allow the additions to complete.

Since the time required by the ripple carry adders determines the overall system clock speed, Spice analysis was performed on the adders. The testing was performed using  $0.8\ \mu\text{m}$  CMOS technology data. A 24-bit adder was tested for maximum propagation delay through the carry path.<sup>3</sup> The total computed delay was then divided by 24, yielding the average delay per bit. The test was repeated at varying operating voltages and the results are illustrated in Figure 46.

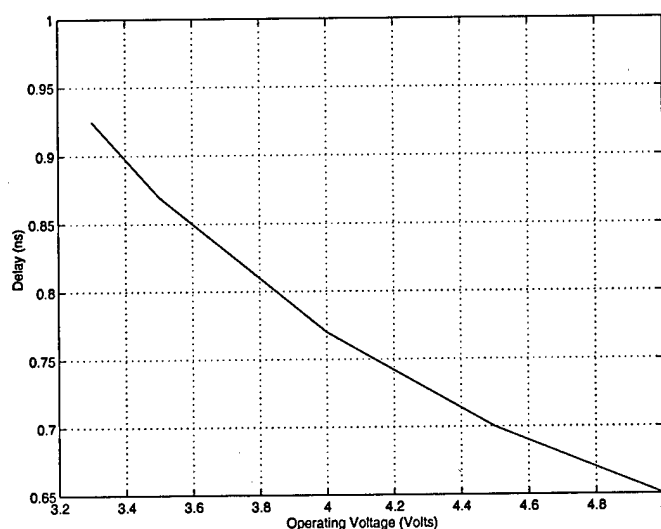


Figure 46: Propagation Delay for Full-Adder Cells

To evaluate speed, consider the data presented in Figure 46. If the operational voltage for the system is 3.3 V, each full adder cell inserts a 0.925 ns delay into the carry path. Therefore, the 26-bit adder required by the filter multiplier would need 24.05 ns to complete its computation. In order for multiplier to operate, the total clock period must be at least 24.05 ns, limiting the system

<sup>3</sup>A 24-bit adder was tested before the length of the longest adder needed was known to establish the delay/bit.

clock to rates under 41.5 MHz. Similarly, if the system is operated at 5.0 V, the minimum delay is 16.9 ns which limits the clock rate to 59.1 MHz.

The filter stage, which requires the most computational cycles, requires 25 clock cycles for the IIR filter. Further, the first instance of the APGF requires 46 clock cycles, while each additional APGF filter requires 42 cycles. Therefore, the proposed 32 channel system requires 1373 clock cycles to process each input value through all channels of the filtering stage.

If the data arrives to the system with a sampling period of  $50\ \mu s$  (20 k-samples/sec), the minimum processing clock period must be no greater than 36.4 ns ( $50\ \mu s / 1373\text{cycles}$ ). Therefore, the architecture must be clocked at a minimum of 27.5 MHz for sustained real-time operation on 32 channels. Since the required minimum clocking frequency for real-time operation is lower than the predicted maximum clock frequency of the hardware, the proposed system is capable of functioning in real time with 32 channels functioning.

#### 5.4 Size

Having shown that the designed architecture meets functional and speed requirements, the next parameter to evaluate is the physical size of the architecture. This section presents the area requirements of the sub-components and produces an estimate for the overall area required for the system. The unit of measure used to compute the area is  $\lambda$ , which represents the smallest drawn line width in an integrated circuit layout. Typically  $\lambda$  is one-half of the smallest actual circuit feature width. For example, for a  $0.8\ \mu m$  fabrication,  $\lambda$  is  $0.4\ \mu m$ . The cells for the VLSI layout were designed according to the MOSIS sub-micron fabrication rules, and remain scalable as technology permits.

Table 12 summarizes the area estimates for the cells of the proposed architecture. In Table 12, the dimensions for most cells were acquired from CMOS VLSI layouts of the cells. In some cases, such as the basic logic gates and the logic for the PLAs, the sizes are an average of typical logic gates. The table is broken into the same three subdivisions that were presented in the discussion of the architecture in Chapter 4: filtering, amplitude compression, and neural encoding.

While Table 12 does include all wiring (power, ground, and clocking) internal to the cells; inter-cell wiring is not included. Inter-cell wiring, which depends heavily upon the technology implemented and layout style, is more uncertain than the size estimations for the functional units.

Estimates vary as to what fraction of a VLSI circuit is devoted to wiring. Additionally, the cost of wiring depends upon the number of layers of interconnect available in the fabrication process being used. Therefore, the following evaluation is based on the assumption that the wiring will "dominate"[31] the total chip area.

Analysis of the fully-custom logarithmic circuit layout revealed the inter-cell wiring consumed nearly 40% of the total circuit area (three levels of interconnect were employed). Using the wiring of the logarithmic circuit as a guide, assume conservatively the wiring for the total system will require an area equal to the functional area, the total area for the circuit would be  $37968776 \lambda^2$ . The estimated area would fit inside a square that measured less than  $6200\lambda$  on a side. If a more conservative estimate is used where the wiring is allowed to occupy two-thirds of the chip area, the total chip would require  $56953164 \lambda^2$ . Under the second and more conservative assumption, the required area would be a square of under  $7600 \lambda$  per side.

In either case, a bonding pad frame is also required. Based upon the sizes for available standard bonding pads, an additional  $150 \mu\text{m}$  is required per side of the wire for bonding. Combining these figures, the resulting integrated circuit would easily fit onto existing fabrication dies available through the Metal Oxide Semiconductor Implementation Service (MOSIS). For example, using the MOSIS  $0.8 \mu\text{m}$  CMOS fabrication process, the more conservative (two-thirds) estimate would require a chip of  $3.3 \text{ mm}$  on a side. If the MOSIS  $0.25 \mu\text{m}$  process were used, the area required by the same chip would be a mere  $1.25 \text{ mm}$  per side.<sup>4</sup>

### 5.5 Power

The last evaluation performed on the proposed architecture was an estimation of the power required for its operation. Power analysis of an integrated circuit depends on many parameters, several of which will remain unknown until the circuit is actually implemented. Some of the unknown parameters included inter-cell bus capacitances, clocking frequency, fabrication technology,

---

<sup>4</sup>MOSIS estimates that the  $0.25 \mu\text{m}$  process be available in the fourth quarter of 1999.

Table 12: Area Estimates For Architecture ( $\lambda$ )

Filtering			
Cell	# Required	Size Each ( $\lambda$ )	Area ( $\lambda^2$ )
Multiplier slice	78	640x42	2,096,640
Mult. Cont	1	72x137	9,864
ROM (34x16)	3	276x168	139,104
RWM (130x24)	2	6565x648	8,508,240
RWM write amp	48	100x27	129,600
RWM read amp	48	100x27	129,600
Adder	26	133x42	145,236
D-Latch	109	103x42	471,534
Transp. Latch	16	54x42	36,288
D-Latch w/ clr	24	103x42	103,824
Clock Gen	5	100x120	60,000
Mux	150	50x75	562,500
Rowselect	164	103x42	709,464
Rowmux	130	50x75	487,500
Logic gates	6	50x50	75,000
PLA	50	50x50	125,000
Amplitude Compression			
Log unit	1	425x640	272,000
Adder	48	133x42	268,128
Latch	20	133x42	111,720
Logic gates	2	50x50	5,000
Neural Encoding			
RWM (32x16)	3	1616x432	2,094,336
RWM write amp	48	100x27	129,600
RWM read amp	48	100x27	129,600
ROM (32x8)	1	260x84	21,840
ROM (32x11)	1	260x116	30,160
Rowselect	32	103x42	138,432
Mux	192	50x75	720,000
Clock gen	3	100x120	36,000
ALU	17	300x42	214,200
D-Latch	43	103x42	186,018
Tranp. Latch	34	54x42	74,256
Multiplier slice	18	640x42	483,840
Mult. Cont	1	72x137	9,864
Logic gates	9	50x50	70,000
PLA	80	50x50	200,000
TOTAL			18,984,388

and operating voltage. These unknown parameters cannot be quantified until the entire architecture has been laid out for a fabrication targeted to a specific technology (or scale). Therefore, the power analysis in presented here assumed fabrication in  $0.8\ \mu m$  CMOS operating at  $33\ MHz$  from a  $5\ V$  power source.

The analysis was accomplished by running Spice simulations on the major functional units to determine their operating power requirements. These estimates were then combined with the utilization rates for the functional units to derive an estimate of the total system's power consumption. The functional units characterized were: multipliers, adders, latches, and state machines. Data from the independent tests were then combined to gain insight into the overall power requirement. The first functional unit to be characterized was the multiplier.

**5.5.1 Multipliers.** To evaluate the multiplier, a test circuit was constructed using the layout of the multiplier slice cells. In order to reduce the computational time of Spice, only six cells were used along with the multiplier controller. The circuit was extracted for Spice using  $0.8\ \mu m$  technology data and five different tests were run. For each test the average current was computed by integrating the instantaneous current and dividing by the length of the simulation. Separate power supplies were simulated for the controller and the array of multiplier cells to isolate the power consumed by each. Table 13 summarizes the results of these tests.

Table 13: Spice Simulation Results for 6-bit Multiplier

Data	Average Current ( $\mu A$ )	
	Control	6 Slice Cells
$23 \times (-9)$	110	320
$(-9) \times 23$	74	215
$(-1) \times (-1)$	75	120
$31 \times 31$	90	200
$21 \times (-22)$	30	240

Examining the data from Table 13, it is easy to see that the power required for the multiplier is highly dependent upon the data stream. The values used in the Spice testing of the 6-bit multiplier were chosen to be representative of various bit patterns that would be encountered to provide a more accurate estimate of the average current required. Using Table 13 the average current for the



multiplier controller is approximately  $76 \mu A$ . Similarly, the average current for the six multiplier slice cells is  $219 \mu A$ , or approximately  $37 \mu A/\text{cell}$ .

Combining the average current of the controller with that of the multiplier slice cells, it can be estimated that the 26-bit multipliers used in the filterbank will consume an average of  $1038 \mu A$ . From the simulated current draw, the average power can be calculated to be  $5.19 \text{ mW}$  when operated continuously. However, the multipliers in the filterbank are only operational in 75.9% of the states. Therefore, a more accurate estimate of the power consumed by each multiplier in the filterbank is  $3.94 \text{ mW}$ . The three multipliers of the filterbank will therefore consume a combined power of approximately  $11.82 \text{ mW}$ .

**5.5.2 Adders.** The second computational cell in the filterbank is the 26-bit adder, which accumulates the multiplier results. From the filterbank state table it can be shown that the 26-bit adder will potentially see new input values in 46.4% of the filterbank's operational states. To estimate the power required by the 26-bit adder, a test similar to that of the multiplier was performed. The simulations revealed that the current consumed per adder cell ranged from zero to a maximum of  $14.8 \mu A/\text{cell}$  depending on the data. The average value ( $7.4 \mu A/\text{cell}$ ) was chosen for analysis. Accounting for the utilization rate of the filterbank adder, the 26-cell device will consume an estimated average of  $446 \mu W$ .

Since the adders depend upon a ripple carry, there will be some additional switching activity where cells may transition more than one time for a computation to complete. Since the switching will only involve the carry path, at most one half of the logic inside each adder will be affected. Also, the added activity will affect on average only half of the cells of the adder. Therefore, a more accurate estimate for the average power consumed by the 26 cell adder is

$$446 \mu W + (446 \mu W) \times (0.5 \text{ gates/cell} \times 0.5 \text{ activity/gate}) = 558 \mu W \quad (29)$$

**5.5.3 Latches.** The data latches on the output of the adder and multiplier were the next cells to be considered. To analyze the latches, a string of 26 cells was assembled for simulation. To simulate the fact that not every bit in a data path changes state on every machine cycle, only half

of the cells were forced to change states with every clock state. The remaining cells were clocked, but not transitioned.

The Spice testing showed that the average current drawn by the 26 latches was  $120\ \mu A$ , or  $4.62\ \mu A$  per cell. Converting the current to power, each cell consumed an average of  $23.1\ \mu W$ . In the filterbank there are 160 latch cells outside of the multipliers which are grouped into 7 registers averaging 22.8 bits in length. Therefore, the average register will consume 0.527 mW during continuous cycling with half of its bits changing state on each cycle.

During the operation of the filterbank, 15 latch cycles are issued for the IIR filter, and 27 latch cycles are issued on behalf of each of the APGF. Although these latch cycles are distributed among the 7 registers, for analysis all of these cycles were assigned to the average 22.8-bit register. The average latch then has a utilization rate of 64.0%, and its estimated power consumption is approximately 0.34 mW.

*5.5.4 State Machines.* The finite-state machine (FSM) has not been designed for VLSI implementation, and therefore cannot be simulated for evaluation. To approximate the power consumed by the FSM, note from Table 12 that the estimated area of the programmable logic array (PLA) is approximately the same size as the 26-bit adder in the filterbank. Although the PLA is smaller than the adder, the difference is offset by the additional gates and counters (latches) that are nested inside the FSM. Therefore, it is reasonable to use the power consumed by the 26-bit adder as an approximation of the power consumed by the FSM.

When cycled on a continuous basis, the adders were simulated to consume a maximum  $14.8\ \mu A$  per cell, or  $384.8\ \mu A$  for the 26-bit adder. Since there is a high level of activity in the FSM due to the constantly changing state, the maximum current drawn by the adder was used to approximate the average current drawn by the FSM. Therefore, the estimated average power consumed by the FSM is 1.92 mW.

*5.5.5 Filterbank Power.* Combined, the multipliers, data latches, adder, and FSM consume an average of 14.6 mW of electrical power. The remaining cells in the filterbank architecture will not significantly contribute to the power consumption of the system due to the in-frequency of their operation and the use of low-power transmission-gate (pass) logic.

**5.5.6 Logarithmic Compression.** The power consumed by the logarithmic compression stage is a negligible portion of the total power consumed by the system. Simulations indicated that the logarithmic conversion process consumes  $450 \mu W$  when cycled continuously at 50 MHz. As implemented in the proposed system, the data rate is less than 1/32 of the system clock rate. Using the same 33 MHz clock rate that was used for the other Spice simulations, the logarithmic data rate is approximately 1 MHz, or 1/50 of the rate used to characterize the logarithmic unit. Therefore, the estimated power consumed by the logarithm approximator is only  $9 \mu W$ .

**5.5.7 Neural Encoding Processor.** The final stage of the architecture is the neural encoding processor (NEP). The NEP is comprised of the same cells used in the filterbank, therefore, the NEP power requirement is estimated using the same technique described for the filterbank. The power analysis of the NEP are summarized in Table 14. The estimated size of the FSM in the NEP is 1.6 times larger than the FSM used in the filterbank. Since the switching activity of the NEP's FSM is very similar to that of the filterbank FSM, the power consumed by the NEP's FSM is estimated to be 1.6 times greater than the estimated power consumption of the filterbank's FSM.

Table 14: Power Analysis for Neural Encoding Processor

Cell	Power/bit	# Bits	Utilization	Total Power
Multiplier	$37 \mu W$	18	63%	$419 \mu W$
ALU	$7.4 \mu W$	17	19.5%	$24.5 \mu W$
Latches	$4.62 \mu W$	16	39%	$23.8 \mu W$
FSM	$1.92 \mu W$	1.6	100%	3.1 mW

The estimated total power required for the NEP is 3.57 mW. Combining the power required by the NEP with that of the filterbank the estimated total power required for the proposed system is 18.2 mW. The total power consumption is small enough to allow the system to be operated from batteries for an extended period.

**5.5.8 Output Buffers and Clocking.** A final consideration for the power analysis is the power consumed by the input/output buffers and the clocking circuits. The power consumed by the output buffers is highly dependent upon the system into which the chip is installed, the length of wires, and the operating voltages. Fortunately, the data rate at the output is not the same as

the internal clocking rate. Rather, data departs the circuit at an average of 32 times the input data rate. Even if the CD sampling rate of 44 kHz is assumed, the output data rate is only 1.4 MHz. Therefore, the power consumed by the output drivers is expected to remain low.

Another power consideration is the internal clocking circuitry. The clocking for the circuit is highly de-centralized. While there is a global clock reference, this clock signal does not directly drive most of the circuitry. Rather, the reference clock is qualified (via simple gate logic) by control signals from the state machines. These qualified clock signals are used to generate local two-phase clock signals where needed.

Because the local clocks are only driven upon demand from the state machine, most of the clocks are idle at any given time. Since the localized clock drivers spend the majority of the time in the disabled state, their average power consumption will be low. Therefore, a detailed analysis of these clock circuits was not performed.

## 5.6 Summary

The analysis presented in this chapter confirm the viability of the proposed algorithms and architecture. Specifically, the algorithms were shown to produce neural activity patterns (NAPs) which did not significantly change the results of phoneme recognition experiments. Further, when the algorithms are implemented using the proposed VLSI architecture, they may be realized in a system that can perform the computation of a 32-channel NAP in real time.

The proposed VLSI architecture is small, estimated to fit inside a square of  $7600 \lambda$  on a side. With these dimensions the processor could be fabricated in any of the technologies currently available through MOSIS. For example, if the MOSIS  $0.8 \mu\text{m}$  CMOS technology were used, the resulting chip would be approximately a square  $3.3 \text{ mm}$  on a side and consume an estimated  $18.2 \text{ mW}$ , making battery operation practical.

If a technology with smaller minimum feature sizes than  $0.8 \mu\text{m}$  were used to implement the system, the area of the die would decrease approximately proportional to the square of the change in feature size. Further, additional layers of metal interconnect which are available on the newer technology processes would also lead to a reduction in size. In terms of power, the smaller feature size fabrication processes are designed for lower operating voltages. By being smaller and thus

having lower drive requirements, these processes are inherently faster and consume less power than the technology used as an example in this chapter. Therefore, the conclusion can be made that the operating characteristics estimated in this chapter for speed, size, and power are bounding cases or worst case values.

## *VI. Conclusions and Recommendations*

### *6.1 Introduction*

The primary objective of this research was to develop an approximation of the Auditory Image Model (AIM) for implementation into VLSI. The modifications to the model needed to be made without significantly distorting its output, the neural activity pattern (NAP). Section 6.2 draws conclusions concerning the modifications implemented, while Section 6.3 proposes recommendations for future research. The final section provides an overall summary of the work.

### *6.2 Conclusions*

Until now, researchers were restricted to the use of workstation computers for the execution of physiologically based auditory models. While these models which mathematically describe the transformations of ear are accurate, they are time consuming to operate. In addition, the processing requirements of such models have prevented them from advancing from the research laboratory into the realm of application. The Auditory Image Model (AIM), which includes several different algorithms for each of the stages of auditory processing is one such model that is restricted by its computational needs.

As demonstrated in Chapter 3, the detailed algorithms in AIM can be approximated using algorithms which require significantly less computation. The approximated algorithms were designed to preserve as much of the original behavior as possible, while providing the maximum reduction in computational cost. As discussed in Chapter 5, the proposed changes to the algorithms in AIM did not alter the overall performance of the model, when used for phoneme recognition, with statistical significance. The result is a new auditory model similar to AIM, but demanding less computation.

The primary advantage of the new auditory model is that its algorithms may be mapped directly onto a relatively simple, single-chip VLSI processor. One possible implementation of the new model was outlined in Chapter 4. The proposed architecture was shown to easily fit onto a square die of 3 mm on each side. Further, it was shown that the processor is capable of sustained real-time operation for 32 channels and requires less than 20 mW of power.

The transformation of AIM from desktop workstations to a low-power integrated circuit opens new doors of opportunity for researchers and will free the auditory model from the confines of the laboratory. The real-time operation of the hardware implementation will allow researchers to more efficiently utilize the model to process audio data for further experimentation. With less time spent on generating data, more time can be spent on analysis and developing new areas of research and application.

Further, because of its small physical size and low power requirement, the integrated circuit implementation of the auditory model may be embedded into other systems. For example, the auditory model could be used as a pre-processor for a speaker identification security system or a real-time voice-to-text system.

The integrated circuit could also allow auditory modeling to move from computational research into the field of medicine. The architecture proposed in Chapter 4 is small enough, and requires so little power that the circuit could be fabricated as a cochlear implant. As an implant, the circuit could be used in individuals with middle and inner ear damage to directly convert sound energy into the electrical neural activity patterns (NAP). The electrical NAP would then be used to stimulate the hair cells of the cochlea and thereby restore hearing. Since the effects of the ear canal were not included as a part of the modified model, the processor and microphone could be implanted allowing the natural resonance of the ear canal to occur.

The search for more efficient algorithms to simplify AIM led to the development of a new technique for the generation of approximate logarithms. While the new technique is an approximation, it produces logarithms with peak errors of less than 1.5% for all numbers, and less than 0.5% for values greater than 25. Since the dynamic range of the input data to the logarithmic unit in the present application is 0 to 32767, the error introduced by the approximate logarithm is expected to be minimal. Additionally, the approximation technique requires only combinational logic, making it very fast and low in power consumption.

The new technique for computing logarithms has application in signal processing (such as AIM) where it may be used in algorithms requiring signal compression. Additionally the technique may be used to reduce computational workloads of other algorithms by mapping multiplication operations into addition operations.

### 6.3 *Recommendations for Further Research*

While the primary objective of transforming AIM was met, areas for future work and research exist. The subsections that follow bring to light some topics to which future research could be directed.

**6.3.1 *Layout Completion.*** The architecture described in Chapter 4 was supported by the VLSI layout of its major cells. Development of these cells was essential to the estimation of power and size, however, the layout of the architecture as a whole has not been completed. Before such a layout could be realized, the finite-state machine logic for the filterbank and neural encoding processor must be constructed. The required equations, which need to be programmed, are tabulated in Appendices B and C. The area and power estimations of Chapter 5 were based upon these equations being implemented by, but not restricted to, a programmable logic array (PLA).

With the generation of the control logic, the connection of the existing cells could be completed. Upon completion of a layout for the architecture, an appropriate interface must be designed. Following fabrication, testing would be required to validate the power and speed estimations.

**6.3.2 *Integrated Circuit Interface.*** Although the architecture discussed in the preceding chapters is complete computationally, one issue not addressed was the incorporation of the circuit into a larger system. Interface issues cannot be addressed until a host system is known. For example, the interface to a computer will be different from an interface to the human ear.

In order to best serve the auditory research community, an interface to a common bus architecture would be desirable. A suggested bus is the Peripheral Component Interconnect (PCI) bus common to most workstations. By including the interface logic for such a bus on-chip, the resulting circuit could easily connect to a workstation and serve as a co-processor for the production of neural activity patterns under the control of the host processor.

The addition of an analog input channel to the architecture would also be advantageous. By incorporating an analog-to-digital converter, as well as the digital bus interface, additional flexibility would be added allowing researchers to process audio signals, such as speech, in real



time. The inclusion of the analog input circuitry would be required in order for the integrated circuit to be utilized as a cochlear implant.

An additional requirement for the circuit to be implantable would be an analog output stage. The analog output stage would require one output channel for each filter in the filterbank. For the architecture described in Chapter 4, thirty-two output channels would be required. However, since the output data is time-division multiplexed, a single digital-to-analog converter could be implemented with one analog hold amplifier dedicated to each output channel.

*6.3.3 Writable Coefficients Store.* Another area of future work is the addition of writable memories for the storing of the coefficients. For the proposed architecture, read-only memories (ROMs) were used for their ease of implementation. The use of ROM's as proposed requires that the coefficients be programmed before the masks for the fabrication are generated. Therefore, the coefficients cannot be altered after fabrication. If new filter frequencies are desired, new fabrication masks are required thus increasing the cost per chip. Additionally, the generation of new custom chips is typically a several month process.

If erasable, programmable ROMs (EPROMs) were used to replace the ROMs in the design, the end user could program the filterbank and NEP coefficients after fabrication. Additionally, the coefficients could be erased and re-programmed if necessary. While there are benefits to programmable coefficient stores, the inclusion of additional logic to allow for the writing and possible erasing of the stores would be necessary.

*6.3.4 Neural Encoding Approximations.* Finally, it was shown through experimentation that the changes made to the algorithms of the neural encoding had the largest impact on the NAP's generated by the model. Further research should be directed toward improving upon these approximations.

One approximation, in particular, that should be studied is the rounding of the neural firing threshold. The approximation used rounded this limit from 1397 to 1024 because of the binary simplicity of 1024. The absolute limit could be raised to 1280 by including only one more bit into the the value, resulting in one more addition. Including one more bit (and thus addition) the threshold could be raised to 1408. By increasing the lower threshold of the nerves sensitivity, the

resulting NAP will have less magnitude error and will not contain some of the smaller features not found in the NAP generated by AIM. The tradeoff between added computation and the resulting accuracy of the NAP requires further study.

#### 6.4 *Summary*

As a result of this research, we are now one step closer to transforming the Auditory Image model into application-specific devices. Algorithms were presented which may be used to significantly simplify the computational workload of the Auditory Image Model. Further, an architecture was designed for an application-specific integrated circuit to execute the new algorithms. The next step will be to complete the layout of the architecture for fabrication and testing.

### Appendix A. Computation of All-Pole Gammatone Filter Constants

This appendix describes the derivation and computation of the constants used for the all-pole gammatone filter replacement. The derivation begins with the S domain expression for the second order stage of the recursive all-pole filter:

$$GT(s) = \frac{1}{(s + B)^2 + \omega_o^2} \quad (30)$$

Where  $B$  is a bandwidth term defined by:

$$B = (2\pi)1.019ERB(f_o) \quad (31)$$

and  $ERB(f)$  is a function to compute the "equivalent rectangular bandwidth" of the gammatone filter at the frequency  $f$ . There exists more than one function to compute  $ERB(f)$ , however, the equation seen most often in recent literature and recommended by Glasberg and Moore [20] is:

$$ERB(f) = 24.7 \left( \frac{4.73f}{1000} + 1 \right) \quad (32)$$

The first step in obtaining the filter coefficients is to express  $GT(s)$  in terms of a discrete equation. Slaney performs this transformation with the aid of computer aided symbolic manipulation to get an expression of the form[17:p. 27]:

$$H(z) = \frac{-2TZ \sin(\omega_o T)}{e^{BT} \left( \frac{-2\omega_o}{e^{2BT}} - 2\omega_o Z^2 + \frac{4\omega_o Z \cos(\omega_o T)}{e^{BT}} \right)} \quad (33)$$

where  $T$  represents the sampling period for the discrete data, and  $B$  and  $\omega_o$  are as defined above. Equation 33 can be simplified to the form:

$$H(z) = \frac{\frac{1}{\omega_o} e^{-BT} T \sin(\omega_o T) Z^{-1}}{1 - 2Z^{-1} e^{-BT} \cos(\omega_o T) + e^{-2BT} Z^{-2}} = \frac{Y(z)}{X(z)} \quad (34)$$

At this point it is critical to point out that the above equations were based on the derivations done by Slaney who normalized the equations so that the peak of all of his filter plots would reach a unity gain (0 dB). To proceed it is important to re-introduce the actual midband gain which has been

removed. To re-introduce the actual midband gain, we must return to the original  $GT(s)$  expression and solve for the magnitude of the gain letting  $\omega = \omega_o$ :

$$GT(s) = \frac{1}{(s + B)^2 + \omega_o^2} \quad (35)$$

Letting  $s = j\omega_o$

$$GT(j\omega) = \frac{1}{(j\omega_o + B)^2 + \omega_o^2} = \frac{1}{B^2 + 2Bj\omega_o} \quad (36)$$

Hence the magnitude of the midband gain is:

$$|GT(j\omega)| = \frac{1}{\sqrt{B^4 + 4B^2\omega_o^2}} = \frac{1}{B\sqrt{B^2 + 4\omega_o^2}} \equiv K \quad (37)$$

We now return to the expression for the Z transformed all-pole gammatone filter and simply divide  $H(z)$  by  $K$  to provide the needed gain at the midband. Finally, solving the resulting expression for  $Y(z)$  and replacng the  $X(z)Z^{-1}$  with  $X_{[n-1]}$  and  $Y(z)Z^{-1}$ ,  $Y(z)Z^{-2}$  with  $Y_{[n-1]}$  and  $Y_{[n-2]}$  respectively, we get:

$$Y_{[n]} = \frac{T}{K\omega_o} e^{-BT} \sin(\omega_o T) X_{[n-1]} + 2e^{-BT} \cos(\omega_o T) Y_{[n-1]} - e^{-2BT} Y_{[n-2]} \quad (38)$$

which is of the form:

$$Y_{[n]} = aX_{[n-1]} + bY_{[n-1]} + cY_{[n-2]} \quad (39)$$

as given in the text of Chapter 4. The constants are then defined by:

$$a = \frac{T}{K\omega_o} e^{-BT} \sin(\omega_o T) \quad (40)$$

$$b = 2e^{-BT} \cos(\omega_o T) \quad (41)$$

$$c = -e^{-2BT} \quad (42)$$

These coefficients can be computed one time and stored as ROM constants for each filter. In the modified AIM code, a compile-time option was added to the file `filter/apgf.c` which when enabled causes AIM to write the these constants to disk files when the program is run.

*Appendix B. State Output Equations for the Filterbank*

$$\begin{aligned}
 ldy &= (sb3 \cdot sb2 \cdot \overline{sb1} \cdot sb0) \\
 adv &= (sb4 \cdot \overline{sb3} \cdot \overline{sb2} \cdot \overline{sb1} \cdot sb0) + (\overline{sb4} \cdot \overline{sb3} \cdot sb2 \cdot \overline{sb1} \cdot sb0) + (sb4 \cdot sb2 \cdot sb1 \cdot \overline{sb0}) + \\
 &\quad (sb3 \cdot sb2 \cdot \overline{sb1} \cdot sb0) \\
 avr &= (\overline{sb4} \cdot \overline{sb3} \cdot sb2 \cdot \overline{sb1} \cdot sb0) + (sb4 \cdot sb2 \cdot sb1 \cdot \overline{sb0}) + (sb3 \cdot sb2 \cdot \overline{sb1} \cdot sb0) \\
 w1 &= (\overline{sb4} \cdot \overline{sb3} \cdot \overline{sb2} \cdot sb1 \cdot sb0) + (sb4 \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) + (sb3 \cdot sb2 \cdot \overline{sb1} \cdot sb0) + \\
 &\quad (sb4 \cdot sb3 \cdot sb0) \\
 w2 &= (\overline{sb4} \cdot \overline{sb3} \cdot \overline{sb2} \cdot sb1 \cdot sb0) + (sb4 \cdot \overline{sb2} \cdot sb1 \cdot sb0) + (sb3 \cdot sb2 \cdot sb0) + (sb4 \cdot sb3 \cdot \overline{sb0}) \\
 wbk &= (sb4 \cdot sb2 \cdot \overline{sb1}) + (sb4 \cdot sb3 \cdot sb0) \\
 ld &= (\overline{sb3} \cdot \overline{sb2} \cdot sb1 \cdot \overline{sb0}) + (\overline{sb4} \cdot sb2 \cdot sb1 \cdot \overline{sb0}) + (sb4 \cdot sb2 \cdot sb1 \cdot sb0) \\
 mpy &= (\overline{sb4} \cdot \overline{sb3} \cdot \overline{sb2} \cdot sb1 \cdot sb0) + (sb3 \cdot \overline{sb2} \cdot \overline{sb1} \cdot \overline{sb0}) + (\overline{sb3} \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) + \\
 &\quad (\overline{sb4} \cdot sb2 \cdot sb1 \cdot sb0) + (sb4 \cdot \overline{sb1} \cdot \overline{sb0}) + (sb4 \cdot sb2 \cdot \overline{sb1}) + (sb4 \cdot sb3 \cdot sb0) \\
 ldr &= (\overline{sb4} \cdot sb3 \cdot \overline{sb2} \cdot \overline{sb1} \cdot sb0) + (sb4 \cdot \overline{sb3} \cdot \overline{sb2} \cdot \overline{sb1} \cdot sb0) + (\overline{sb4} \cdot \overline{sb3} \cdot sb2 \cdot \overline{sb1} \cdot sb0) + \\
 &\quad (sb4 \cdot sb2 \cdot sb1 \cdot \overline{sb0}) \\
 s0 &= (sb4 \cdot \overline{sb2} \cdot sb1) \\
 s10 &= (\overline{sb4} \cdot sb2 \cdot sb1 \cdot sb0) + (\overline{sb2} \cdot sb0) + (sb4 \cdot sb3 \cdot \overline{sb0}) \\
 s11 &= (sb3 \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) \\
 s2 &= (sb4 \cdot \overline{sb2} \cdot sb1 \cdot sb0) + (sb3 \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) + (\overline{sb4} \cdot sb2 \cdot sb1 \cdot sb0) + \\
 &\quad (sb3 \cdot \overline{sb2} \cdot sb1) + (sb4 \cdot sb3 \cdot \overline{sb0}) \\
 s3 &= (\overline{sb4} \cdot sb2 \cdot sb1 \cdot \overline{sb0}) + (sb4 \cdot sb2 \cdot sb1 \cdot sb0) + (sb3 \cdot sb2 \cdot \overline{sb1} \cdot sb0) \\
 s4 &= (sb4 \cdot sb2 \cdot \overline{sb1}) + (sb4 \cdot sb3 \cdot sb0) \\
 adc &= (sb4 \cdot \overline{sb2} \cdot sb1 \cdot sb0) + (sb3 \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) + (\overline{sb4} \cdot sb2 \cdot sb1 \cdot sb0) + \\
 &\quad (sb3 \cdot \overline{sb2} \cdot sb1) + (sb4 \cdot sb3 \cdot \overline{sb0}) \\
 ldo &= (\overline{sb4} \cdot \overline{sb3} \cdot sb2 \cdot sb1) + (sb4 \cdot sb2 \cdot sb1 \cdot sb0) + (sb3 \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) + \\
 &\quad (sb4 \cdot \overline{sb2} \cdot sb1) + (sb3 \cdot \overline{sb2} \cdot sb1) + (sb4 \cdot sb3 \cdot \overline{sb0}) \\
 cpl &= (sb4 \cdot sb3 \cdot sb0)
 \end{aligned}$$

### Appendix C. State Output Equations for NEP State Machine

$$\begin{aligned}
 lmp &= (\overline{sb4} \cdot \overline{sb3} \cdot \overline{sb2} \cdot sb1 \cdot \overline{sb0}) + (sb3 \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) + (sb2 \cdot sb1 \cdot sb0) + \\
 &\quad (sb4 \cdot sb1 \cdot sb0) + (sb4 \cdot sb2 \cdot sb1) \\
 s50 &= (sb3 \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) + (sb4 \cdot sb1 \cdot sb0) + (sb4 \cdot sb2 \cdot sb1) \\
 s51 &= (\overline{sb4} \cdot \overline{sb3} \cdot \overline{sb2} \cdot sb1 \cdot \overline{sb0}) \\
 s6 &= (\overline{sb4} \cdot \overline{sb3} \cdot \overline{sb2} \cdot sb1 \cdot \overline{sb0}) + (sb3 \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) + (sb4 \cdot sb1 \cdot sb0) + \\
 &\quad (sb4 \cdot sb2 \cdot sb1) \\
 s7 &= (\overline{sb4} \cdot \overline{sb3} \cdot sb2 \cdot sb1 \cdot \overline{sb0}) + (sb3 \cdot \overline{sb2} \cdot sb1 \cdot sb0) \\
 s8 &= (\overline{sb4} \cdot \overline{sb3} \cdot sb2 \cdot sb1 \cdot \overline{sb0}) + (sb4 \cdot \overline{sb2} \cdot sb1 \cdot \overline{sb0}) + (sb3 \cdot \overline{sb2} \cdot sb1 \cdot sb0) \\
 s9 &= (sb2 \cdot sb1 \cdot sb0) + (sb4 \cdot sb1 \cdot sb0) \\
 s10 &= (sb4 \cdot sb1 \cdot sb0) \\
 mp2 &= (\overline{sb4} \cdot \overline{sb3} \cdot \overline{sb2} \cdot sb1 \cdot sb0) + (sb4 \cdot \overline{sb2} \cdot \overline{sb1} \cdot \overline{sb0}) + (sb3 \cdot \overline{sb2} \cdot \overline{sb1} \cdot \overline{sb0}) + \\
 &\quad (sb3 \cdot sb2 \cdot \overline{sb1} \cdot sb0) \\
 lpf &= (sb3 \cdot sb2 \cdot sb1 \cdot sb0) + (sb3 \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) + (sb4 \cdot sb1 \cdot sb0) + (sb4 \cdot sb2 \cdot sb1) \\
 adj &= (sb3 \cdot \overline{sb2} \cdot \overline{sb1} \cdot sb0) + (sb3 \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) \\
 lr &= (\overline{sb4} \cdot \overline{sb3} \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) + (sb3 \cdot \overline{sb2} \cdot \overline{sb1} \cdot sb0) + (sb3 \cdot sb2 \cdot sb1 \cdot \overline{sb0}) + \\
 &\quad (sb4 \cdot \overline{sb1} \cdot sb0) \\
 as &= (\overline{sb4} \cdot \overline{sb3} \cdot sb2 \cdot sb1 \cdot \overline{sb0}) \\
 lmic &= (\overline{sb3} \cdot sb2 \cdot \overline{sb1} \cdot sb0) + (sb3 \cdot \overline{sb2} \cdot sb1 \cdot sb0) + (sb3 \cdot sb2 \cdot sb1 \cdot sb0) \\
 pas &= (sb3 \cdot \overline{sb2} \cdot sb1 \cdot \overline{sb0}) + (sb3 \cdot sb2 \cdot sb1 \cdot sb0) \\
 ltmp &= (\overline{sb4} \cdot \overline{sb3} \cdot sb2 \cdot sb1 \cdot \overline{sb0}) + (sb4 \cdot \overline{sb2} \cdot sb1 \cdot \overline{sb0}) + (sb3 \cdot \overline{sb2} \cdot sb1 \cdot \overline{sb0}) \\
 wmc &= (sb3 \cdot sb2 \cdot \overline{sb1} \cdot \overline{sb0}) + (sb4 \cdot sb2 \cdot sb1) \\
 wlp &= (sb4 \cdot sb1 \cdot sb0) \\
 zro &= (sb4 \cdot sb2 \cdot sb1) \\
 dorq &= (sb4 \cdot sb2 \cdot \overline{sb1})
 \end{aligned}$$

#### Appendix D. Rom Layout Generation Code

```
#include <stdio.h>
#include <time.h>

#define X 8          /* width of generated cells */
#define Y 16         /* height of generated cells */
#define SPACING 4

int main(int argc, char *argv[])
{
    int cx, cy;
    int val1, val2, i, j, bit1, bit2;
    FILE *fin, *fout;
    time_t t;
    int row, bits, dolabels=1;
    char ofname[64];

    if (argc == 1) {
        fprintf(stderr, "\n\naUSAGE: mkrom [wordsize] datafile\n");
        fprintf(stderr, "\tIf wordsize is not specified, 16 bits is \
            defaulted\n");
        fprintf(stderr, "\tThe datafile must contain one data value per \
            line in integer form.\n");
        fprintf(stderr, "\tIf there are an odd number of values, a row \
            of all 0's will be appended.\n");
        fprintf(stderr, "\n\tThe output file will be datafile.mag\n");
        exit(-1);
    }

    if (argc == 2)
        bits=16;
    else {
        bits = atoi(argv[1]);
        if (bits == 0) {
            fprintf(stderr, "ERROR: word size of zero was specified\n");
            fprintf(stderr, "\n\naUSAGE: mkrom [wordsize] datafile\n");
            fprintf(stderr, "\tIf wordsize is not specified, 16 bits is \
                defaulted\n");
            fprintf(stderr, "\tdatafile must contain one data value per line \
                in integer form.\n");
            exit(-2);
        }
    }

    fin = fopen(argv[argc-1], "r");
    if (fin == NULL) {
        printf("ERROR OPENING INPUT FILE %s\n", argv[argc-1]);
        exit (-1);
    }
}
```

```

sprintf(ofname, "%s.mag", argv[argc-1]);
fout = fopen(ofname, "w");
if (fout == NULL) {
    printf("ERROR OPENING OUTPUT FILE %s\n", ofname);
    exit (-1);
}

t = time(NULL);
/* Write out header */
fprintf(fout, "magic\ntech scmos\ntimestamp %ld\n", t);

/* Write out bottom row ground plane */
cy = 0;
cx = 0;
for (i=0; i<=bits ; i++){ /* extra insures right side ground */
    if (!(i%SPACING) || (i == bits)){
        fprintf(fout, "<< metall >>\n");
        fprintf(fout, "rect %d %d %d %d\n", cx+1, cy, cx+7, cy+4);
        fprintf(fout, "<< ndcontact >>\n");
        fprintf(fout, "rect %d %d %d %d\n", cx+1, cy, cx+7, cy+4);
        fprintf(fout, "<< ndiffusion >>\n");
        fprintf(fout, "rect %d %d %d %d\n", cx, cy, cx+8, cy+4);
        cx+=X;
    }
    if (i < bits ) {
        fprintf(fout, "<< ndiffusion >> \n");
        fprintf(fout, "rect %d %d %d %d\n", cx, cy, cx+X, cy+4);
        fprintf(fout, "<< metall >> \n");
        fprintf(fout, "rect %d %d %d %d\n", cx+2, cy, cx+5, cy+4);
        fprintf(fout, "<< labels >>\n");
        fprintf(fout, "rlabel metall %d %d %d %d 1 B%d\n", \
            cx+3, cy+1, cx+3, cy+1, i);
        cx+=X;
    }
}
}
cy = 4;
row = 0;
/* Write out actual rom data cells */
while (fscanf(fin, "%d", &val1)==1) {
    if (fscanf(fin, "%d", &val2)!=1)
        val2 = 0;
    cx = 0;
    for (i=0; i<=bits ; i++){ /* Write vertical ground lines */
        if (!(i%SPACING) || (i == bits)){
            fprintf(fout, "<< metall >>\n");
            fprintf(fout, "rect %d %d %d %d\n", cx+1, cy, cx+7, cy+16);
            fprintf(fout, "<< ndcontact >>\n");
            fprintf(fout, "rect %d %d %d %d\n", cx+1, cy+12, cx+7, cy+16);
            fprintf(fout, "<< ndiffusion >>\n");
            fprintf(fout, "rect %d %d %d %d\n", cx, cy+12, cx+8, cy+16);
            fprintf(fout, "<< polysilicon >>\n");

```



```

    fprintf(fout,"rect %d %d %d %d\n",cx,cy+9,cx+8,cy+11);
    fprintf(fout,"rect %d %d %d %d\n",cx,cy+1,cx+8,cy+3);
    fprintf(fout,"<< psubstratepcontact >>\n");
    fprintf(fout,"rect %d %d %d %d\n",cx+2,cy+4,cx+6,cy+8);
    if (i == 0) { /* label the row lines */
        fprintf(fout,"<< labels >>\n");
        fprintf(fout,"rlabel polysilicon %d %d %d %d 1 R%d\n",
            cx+1,cy+2,cx+1,cy+2,row);
        fprintf(fout,"rlabel polysilicon %d %d %d %d 1 R%d\n",
            cx+1,cy+10,cx+1,cy+10,row+1);
        row += 2;
    }
    cx+=X;
}
if (i < bits) {
    bit1 = val1 % 2;
    bit2 = val2 % 2;
    val1 >>= 1;
    val2 >>= 1;
    fprintf(fout,"<< metall >>\n");
    fprintf(fout,"rect %d %d %d %d\n",cx+2,cy,cx+5,cy+16);
    fprintf(fout,"<< ndcontact >>\n");
    fprintf(fout,"rect %d %d %d %d\n",cx+2,cy+4,cx+6,cy+8);
    fprintf(fout,"<< polysilicon >>\n");
    fprintf(fout,"rect %d %d %d %d\n",cx,cy+9,cx+8,cy+11);
    fprintf(fout,"rect %d %d %d %d\n",cx,cy+1,cx+8,cy+3);
    fprintf(fout,"<< ndiffusion >>\n");
    fprintf(fout,"rect %d %d %d %d\n",cx,cy+12,cx+8,cy+16);
    if (!bit1)
        fprintf(fout,"rect %d %d %d %d\n",cx+2,cy,cx+6,cy+4);
    if (!bit2)
        fprintf(fout,"rect %d %d %d %d\n",cx+2,cy+8,cx+6,cy+12);
    cx += X;
}
}
cy += Y;
}
fprintf(fout,"<< end >>\n");
return 0;
}

```

## Appendix E. VHDL Code

```
-----
-- FILENAME:  filter.vhd
-- AUTHOR:    Sam L. SanGregory
-- DATE:      1/6/99
-- REVISIONS: none
--
-- STATUS:    FROZEN DO NOT MODIFY (witout comment)
--            6/10/99 Removed a carry feedback path in the adder
--
-- FUNCTION:
--            This code models the structure of the main filter bank
--            for the AIM inner ear.  It includes all of the multipliers,
--            adders, memory, and multiplexors needed to perform the
--            outer/middle-ear filter as well as the cochlear filtering.
--
-- DEPENDS UPON:
--            add_n.vhd, fsm_multiplier.vhd, mux2.vhd, mux4.vhd, row_sel.vhd,
--            rowmux.vhd, ram.vhd, rom.vhd, tnreg.vhd, n_dffa1.vhd,
--            n_dff.vhd, andgate.vhd, phi2.vhd
--
-- This source code was written in partial fulfillment of PhD
-- requirements at The Air Force Institute of Technology.
-----

library ieee;
use ieee.std_logic_1164.all;

-----
--          ENTITY          --
-----

entity AIMfilter is
    generic (rom1  : string := "rom1.data";
             rom2  : string := "rom2.data";
             rom3  : string := "rom3.data";
             rom_w : natural := 16;           -- width of rom data
             rom_n : natural := 12;           -- number of rom rows (channels+2)
             ram_w : natural := 24;           -- width of ram data
             ram_n : natural := 42;           -- number of ram rows (4*chan)+2
    );
    port (x      : in  std_ulogic_vector(15 downto 0); -- input data
          clk    : in  std_ulogic;                   -- system clock
          drq    : in  std_ulogic;                   -- input data ready
          dxak   : in  std_ulogic;                   -- acknowledge from next
          reset  : in  std_ulogic;                   -- system reset

          y      : out std_ulogic_vector(15 downto 0); -- output data
          dak    : out std_ulogic;                   -- filter busy
          dxrq   : out std_ulogic;                   -- data ready for next
    );
end AIMfilter;

-----
--          ARCHITECTURE    --
-----
```

-----  
architecture str of AIMfilter is

component add\_n

```
    generic (n : natural := 8;
             td : time := 1 ns);
    port(ai, bi : in std_ulogic_vector(n-1 downto 0);
          ci : in std_ulogic;
          sum : out std_ulogic_vector(n-1 downto 0);
          co : out std_ulogic);
```

end component;

component fsm

```
    generic (td : time := 1 ns);
    port (ck : in std_ulogic;           -- clock input
          drq : in std_ulogic;         -- data request
          rx : in std_ulogic;          -- RWM row X
          reset : in std_ulogic;       -- reset
          dxak : in std_ulogic;

          dak : out std_ulogic;
          ldy : out std_ulogic;
          adv : out std_ulogic;
          avr : out std_ulogic;
          w1 : out std_ulogic;
          w2 : out std_ulogic;
          wbk : out std_ulogic;
          ld : out std_ulogic;
          mpy : out std_ulogic;
          ldr : out std_ulogic;
          s0 : out std_ulogic;
          s10 : out std_ulogic;
          s11 : out std_ulogic;
          s2 : out std_ulogic;
          s3 : out std_ulogic;
          s4 : out std_ulogic;
          ldo : out std_ulogic;
          cpl : out std_ulogic;
          dxrq : out std_ulogic;
          rst : out std_ulogic);
```

end component;

component multiplier

```
    generic (w : natural := ram_w+2;
             cont_td : time := 1 ns;
             tdff_td : time := 1 ns;
             addsub_td : time := 1 ns;
             mux2_td : time := 1 ns;
             dffcl_td : time := 1 ns;
             dff2_td : time := 1 ns);
    port (m1,m2 : in std_ulogic_vector(w-1 downto 0);
          m1out : out std_ulogic_vector(w-1 downto 0);
          c, cb : in std_ulogic;
          ld, ldb : in std_ulogic);
```

```

        result      : out std_ulogic_vector(2*w-1 downto 0));
end component;

component mux2
  generic (td : time      := 1 ns;
           w  : natural := 8  );
  port (i0, i1 : in  std_ulogic_vector(w-1 downto 0);
        s      : in  std_ulogic;
        o1     : out std_ulogic_vector(w-1 downto 0));
end component;

component mux4
  generic (td : time      := 1 ns;
           w  : natural := 8  );
  port (i3, i2, i1, i0 : in  std_ulogic_vector(w-1 downto 0);
        s1, s0         : in  std_ulogic;
        o1             : out std_ulogic_vector(w-1 downto 0));
end component;

component row_sel
  generic (n : natural := 32; td : time := 1 ns);
  port(reset : in  std_ulogic;
        adv  : in  std_ulogic;
        rows : out std_ulogic_vector(0 to n-1);
        rx   : out std_ulogic
  );
end component;

component rowmux
  generic (n : natural := ram_n; td : time := 1 ns);
  port (irow : in  std_ulogic_vector(n downto 0);
        back : in  std_ulogic;
        orow : out std_ulogic_vector(n-1 downto 0)
  );
end component;

component ram
  generic(tdr, tdw : time := 1 ns; -- delay read, write
         w  : natural := ram_w;    -- width of each word
         nw : natural := ram_n;    -- number of words (rows)
         init : natural := 0);     -- initial values for ram
  port (addr : in  std_ulogic_vector(nw-1 downto 0);
        din  : in  std_ulogic_vector(w-1 downto 0);
        wr   : in  std_ulogic;
        dout : out std_ulogic_vector(w-1 downto 0));
end component;

component rom
  generic(tdr: time := 1 ns;      -- delay read
         w  : natural := rom_w;   -- width of each word
         nw : natural := rom_n;   -- number of words (rows)
         fn : string := "romx.data");
  port (addr : in  std_ulogic_vector(nw-1 downto 0);
        dout : out std_ulogic_vector(w-1 downto 0));
end component;

```

```

component tnreg                                -- Transparent Data Latch
  generic (td : time      := 1 ns;    -- propagation delay
          n  : natural := 8 );    -- number of bits
  port (din : in  std_ulogic_vector(n-1 downto 0);
        dout : out std_ulogic_vector(n-1 downto 0);
        load : in  std_ulogic;
        loadb : in  std_ulogic);
end component;

component n_dffacl  -- asynch clear edge triggered
  generic( n : natural := 8;
          td : time := 1 ns);
  port ( d : in  std_ulogic_vector(n-1 downto 0);
        c : in  std_ulogic;
        reset : in  std_ulogic;
        q,qb : out std_ulogic_vector(n-1 downto 0));
end component;

component n_dff                                -- Falling Edge Latch
  generic( n : natural := 8;
          td : time := 1 ns);
  port ( d : in  std_ulogic_vector(n-1 downto 0);
        c : in  std_ulogic;
        q,qb : out std_ulogic_vector(n-1 downto 0));
end component;

component andgate
  generic (n : natural := 2;
          td : time := 1 ns);
  port (i : in  std_ulogic_vector(n-1 downto 0);
        o : out std_ulogic);
end component;

component phi2
  generic (td : time := 1 ns);
  port (ck : in  std_ulogic;
        ck1 : out std_ulogic;    -- in phase with ck
        ck2 : out std_ulogic);  -- out of phase with ck
end component;

signal m0bus : std_ulogic_vector(25 downto 0);  -- MUX outputs
signal m1bus : std_ulogic_vector(25 downto 0);
signal m2bus : std_ulogic_vector(25 downto 0);
signal m3bus : std_ulogic_vector(23 downto 0);
signal m4bus : std_ulogic_vector(23 downto 0);
signal mp1 : std_ulogic_vector(51 downto 0);  -- mult outputs
signal mp2 : std_ulogic_vector(51 downto 0);
signal mp3 : std_ulogic_vector(51 downto 0);
signal mpr1 : std_ulogic_vector(25 downto 0);  -- mult latch outs
signal mpr2 : std_ulogic_vector(25 downto 0);
signal mpr3 : std_ulogic_vector(25 downto 0);
signal abus : std_ulogic_vector(25 downto 0);  -- adder output w/cy
signal dobus : std_ulogic_vector(25 downto 0);  -- adder output latch
signal xbus : std_ulogic_vector(23 downto 0);  -- x latch output
signal ybus : std_ulogic_vector(23 downto 0);  -- y latch output
signal mlout : std_ulogic_vector(25 downto 0);  -- multiplier m1 recycle

```

```

signal r1bus : std_ulogic_vector(25 downto 0); -- ram 1 out
signal r2bus : std_ulogic_vector(25 downto 0); -- ram 2 out
signal rm1 : std_ulogic_vector(25 downto 0); -- rom 1 bus
signal rm2 : std_ulogic_vector(25 downto 0); -- rom 2 bus
signal rm3 : std_ulogic_vector(25 downto 0); -- rom 3 bus
-- ram row selects
signal ramrow : std_ulogic_vector(ram_n-1 downto 0) := (others=>'0');
-- ram1 row selects
signal ram1row: std_ulogic_vector(ram_n-1 downto 0) := (others=>'0');
-- ram rows with rx
signal rxrows: std_ulogic_vector(ram_n downto 0) := (others=>'0');
-- rom row selects
signal romrow : std_ulogic_vector(rom_n-1 downto 0) := (others=>'0');
signal mx1_gnd: std_ulogic_vector(25 downto 0); -- zero input for mux1

signal a1,a2,a3,a4,a5 : std_ulogic_vector(1 downto 0); -- andgate input vect
signal a6,a7 : std_ulogic_vector(1 downto 0); -- andgate input vect
signal mpyi, mpy1, mpy2 : std_ulogic; -- clock for multipliers
signal ldi, ld1, ld2 : std_ulogic; -- load for multipliers
signal ldri, ldr1, ldr2 : std_ulogic; -- load for multiply result
signal xld1, xld2 : std_ulogic; -- load for X register
signal cp1, cp2 : std_ulogic; -- load for compress latch
signal ldock : std_ulogic; -- ldo anded with clock
signal wlck, w2ck : std_ulogic; -- qualified ram writes

signal ldy, adv, avr, w1, w2, wbk : std_ulogic; -- FSM signals
signal ld, mpy, ldr, s0, s10, s11 : std_ulogic;
signal s2, s3, s4, ldo, cpl : std_ulogic;
signal rst, rx : std_ulogic;

signal gnd : std_ulogic := '0';
begin

----- RAM Memory
ramsel : row_sel
generic map( ram_n, 1 ns)
port map( rst, adv, ramrow, rx);

rxrows(ram_n-1 downto 0) <= ramrow;
rxrows(ram_n) <= rx;
ram1mux : rowmux
generic map( ram_n, 1 ns)
port map( rxrows, wbk, ram1row);

a6 <= w1 & clk;
and6 : andgate
generic map (2, 1 ns)
port map (a6,wlck);

ram1 : ram -- using component defaults on generics
port map( ram1row, m4bus, wlck, r1bus(23 downto 0));
r1bus(24) <= r1bus(23); -- sign extension
r1bus(25) <= r1bus(23); -- sign extension

a7 <= w2 & clk;
and7 : andgate

```

```

    generic map (2, 1 ns)
    port map (a7,w2ck);

ram2 : ram                -- using component defaults on generics
    port map( ramrow, m1out(23 downto 0), w2ck, r2bus(23 downto 0));
r2bus(24) <= r2bus(23);    -- sign extension
r2bus(25) <= r2bus(23);    -- sign extension

----- ROM Memory
romaddr : row_sel
    generic map( rom_n, 1 ns)
    port map( rst, avr, romrow, open);
rom_1 : rom
    generic map( 1 ns, rom_w, rom_n, rom1)
    port map( romrow, rm1(15 downto 0));
rm1(rm1'left downto 16) <= (others=>rm1(15));    -- sign extension

rom_2 : rom
    generic map( 1 ns, rom_w, rom_n, rom2)
    port map( romrow, rm2(15 downto 0));
rm2(rm2'left downto 16) <= (others=>rm2(15));    -- sign extension

rom_3 : rom
    generic map( 1 ns, rom_w, rom_n, rom3)
    port map( romrow, rm3(15 downto 0));
rm3(rm3'left downto 16) <= (others=>rm3(15));    -- sign extension

----- Multipliers
a1 <= mpy & clk;
and1 : andgate
    generic map(2, 1 ns)
    port map( a1,mpyi);

a2 <= ld & clk;
and2 : andgate
    generic map(2, 1 ns)
    port map( a2,ldi);

a3 <= ldr & clk;
and3 : andgate
    generic map(2, 1 ns)
    port map( a3,ldr1);

clock1 : phi2
    port map( mpyi, mpy1, mpy2);    -- clock for multipliers

clock2 : phi2
    port map( ldi, ld1, ld2);    -- load for multipliers

clock3 : phi2
    port map( ldr1, ldr1, ldr2);    -- load for mult result

m1t1 : multiplier          -- component default generics
    port map( m0bus, rm1, open, mpy1, mpy2, ld1, ld2, mp1);

m1t2 : multiplier          -- component default generics

```

```

    port map( r1bus, rm2, mlout, mpy1, mpy2, ld1, ld2, mp2);

mlt3 : multiplier      -- component default generics
    port map( r2bus, rm3, open, mpy1, mpy2, ld1, ld2, mp3);

-- The 8 bit offset in the mapping below accounts for the fact that
-- the rom word is 8 bits smaller than that of the ram so multiply
-- is not fully shifted in booth algorithm. Also the range (39-14)
-- accounts for a missing 2 bits of shift. An additional shift
-- operation is required to complete the Booth's but to save a
-- clock cycle this final shift is hardwired on the output.

-- Multiplier output latches
latch1 : tnreg
    generic map( 1 ns, 26)
    port map( mp1(39+8 downto 14+8), mpr1, ldr1, ldr2);

latch2 : tnreg
    generic map( 1 ns, 26)
    port map( mp2(39+8 downto 14+8), mpr2, ldr1, ldr2);

latch3 : tnreg
    generic map( 1 ns, 26)
    port map( mp3(39+8 downto 14+8), mpr3, ldr1, ldr2);

----- Muxes (all)
mx0 : mux2
    generic map( 1 ns, 24)
    port map( m3bus, mlout(23 downto 0), s0, m0bus(23 downto 0));
m0bus(24) <= m0bus(23); -- sign extension
m0bus(25) <= m0bus(23);

mx1_gnd <= (others=>'0'); -- zero unused input
mx1 : mux4
    generic map( 1 ns, 26)
    port map( mx1_gnd, mpr3, mpr2, mpr1, s11, s10, m1bus);

mx2 : mux2
    generic map( 1 ns, 26)
    port map( mpr3, dobus, s2, m2bus);

mx3 : mux2
    generic map( 1 ns, 24)
    port map( xbus, ybus, s3, m3bus);

mx4 : mux2
    generic map( 1 ns, 24)
    port map( m3bus, dobus(25 downto 2), s4, m4bus);

----- X input
clock4 : phi2
    port map( drq, xld1, xld2);

latch4 : tnreg
    generic map( 1 ns, 16)
    port map( x, xbus(23 downto 8), xld1, xld2);

```



```

xbus(7 downto 0) <= (others=>'0');          -- zero pad fraction

----- Y latch
latch5 : n_dffa1
  generic map( 24, 1 ns)
  port map( dobus(24 downto 1), ldy, reset, ybus);

----- Finite State Machine
fsm1 : fsm
  generic map(1 ns)
  port map( clk, drq, rx, reset, dxak, dak, ldy, adv, avr,
            w1, w2, wbk, ld, mpy, ldr, s0, s10, s11,
            s2, s3, s4, ldo, cpl, dxrq, rst);

----- Adder and final output
adder : add_n
  generic map( 26, 1 ns)
  port map( m1bus, m2bus, gnd, abus, open);

a5 <= ldo & clk;
and5 : andgate
  generic map(2, 1 ns)
  port map( a5, ldock);

ndff1 : n_dff
  generic map( 26, 1 ns)
  port map( abus, ldock, dobus, open);

clock6 : phi2
  port map( cpl, cp1, cp2); -- load for mult result

latch6 : tnreg
  generic map( 1 ns, 16)
  -- CHANGED 5/26/99 from (25 downto 10)
  port map( dobus(23 downto 8), y, cp1, cp2);

end str;

-----
-- CONFIGURATION --
-----
configuration AIMfilter_cfg of AIMfilter is
for str
  for all : add_n
    use configuration work.add_n_cfg;
  end for;
  for all : fsm
    use configuration work.fsm_cfg;
  end for;
  for all : multiplier
    use configuration work.multiplier_cfg;
  end for;
  for all : mux2
    use entity work.mux2(beh);
  end for;
  for all : mux4

```

```

        use entity work.mux4(beh);
    end for;
    for all : row_sel
        use configuration work.row_sel_cfg;
    end for;
    for all : rowmux
        use configuration work.rowmux_cfg;
    end for;
    for all : ram
        use entity work.ram(beh);
    end for;
    for all : rom
        use entity work.rom(beh);
    end for;
    for all : tnreg
        use configuration work.tnreg_cfg;
    end for;
    for all : n_dff
        use configuration work.n_dff_cfg;
    end for;
    for all : andgate
        use entity work.andgate(beh);
    end for;
    for all : phi2
        use configuration work.phi2_cfg;
    end for;
    for all : n_dffa1
        use configuration work.n_dffa1_cfg;
    end for;
end for;
end AIMfilter_cfg;

```

```

-----
-- FILENAME:  logapr.vhd
-- AUTHOR:    Sam L. SanGregory
-- DATE:      8/15/98
-- REVISIONS:
--   12/23/98 : Added configuration to test second order approximation
--   1/28/99  : Added shift2 and bits2 to make 2nd approx generic
--
-- STATUS:    FROZEN DO NOT MODIFY (without comment)
-- FUNCTION:
--   This is the top-level of the structural description of the
--   log-base-2 approximator. Two configurations are given,
--   the first is the complete algorithm developed in this research
--   while the second (DUMMY) does not include the adjustment and
--   is thus closer to the original algorithm from WSU.
--
-- DEPENDS UPON:
--   barell.vhd, adjust.vhd, exprom.vhd
--
-- This source code was written in partial fulfillment of PhD
-- requirements at The Air Force Institute of Technology.
-----

```

```

library ieee;
use ieee.std_logic_1164.all;

```

```

-----
-- ENTITY --
-----
entity logapr is
  generic (td : time := 1 ns;
           shift : natural := 2; -- how many positions to shift
           shift2 : natural := 3; -- how many 2nd approx
           bits : natural := 4; -- how many bits to shift
           bits2 : natural := 4); -- how many bits 2nd approx
  port (din : in std_ulogic_vector(15 downto 0);
        eval_not : in std_ulogic;
        logerr : out std_ulogic;
        dout : out std_ulogic_vector(15 downto 0));
end logapr;

```

```

-----
-- ARCHITECTURE --
-----
architecture str of logapr is

  component barell
    generic (td : time := 1 ns);
    port (din : in std_ulogic_vector(15 downto 0);
          eval_not : in std_ulogic;
          logerr : out std_ulogic;
          dout : out std_ulogic_vector(14 downto 0);
          shout : out std_ulogic_vector(14 downto 0));
  end component;

  component adjust

```

```

generic (td : time := 1 ns;
        wid  : natural := 12;
          shift : natural := shift;
          shift2 : natural := shift2;
          bits  : natural := bits;
          bits2 : natural := bits2);
port(x : in std_ulogic_vector(wid-1 downto 0);
     y : out std_ulogic_vector(wid-1 downto 0));
end component;

component exprom
  generic(td : time := 1 ns);
  port(shift : in std_ulogic_vector(14 downto 0);
       log_err : in std_ulogic;
       eval : in std_ulogic;
       exp : out std_ulogic_vector(3 downto 0));
end component;

signal d_tmp1 : std_ulogic_vector(14 downto 0);
signal d_tmp2 : std_ulogic_vector(11 downto 0);
signal shift : std_ulogic_vector(14 downto 0);
signal expv : std_ulogic_vector(3 downto 0);
signal logr : std_ulogic;
signal evl : std_ulogic;

begin

shifter : barell
  port map(din, eval_not, logr, d_tmp1, shift);

adjuster : adjust
  port map(d_tmp1(13 downto 2), d_tmp2);

evl <= not eval_not after 1 ns;
exponent : exprom
  port map(shift, logr, evl, expv);

dout(15 downto 12) <= expv;
dout(11 downto 0) <= d_tmp2;
logerr <= logr;
end str;

-----
--  CONFIGURATION  --
-----
configuration logapr_cfg of logapr is
for str
  for all : barell
    use configuration work.barell_cfg;
  end for;
  for all : adjust
    use configuration work.adjust_cfg;
  -- use entity work.adjust(beh);
  end for;
  for all : exprom
    use entity work.exprom(beh);

```

```
    end for;  
end for;  
end logapr_cfg;
```

```

-----
-- FILENAME:  adaptive.vhd
-- AUTHOR:    Sam L. SanGregory
-- DATE:      5/06/99
-- REVISIONS: none
-- STATUS:    FROZEN DO NOT MODIFY (without comment)
-- FUNCTION:
--   This is the final stage of the ear model. In this stage
--   the adaptive thresholding is accomplished, as well as the
--   final integration filter. This is a structural model.
--
-- DEPENDS UPON: ram.vhd, rom.vhd, rowselect.vhd, mux.vhd, alucell.vhd,
--               2phase_clock.vhd, invert.vhd, andgate.vhd, orgate.vhd,
--               latches.vhd, multiplier.vhd, napfsm.vhd
--
-- This source code was written in partial fulfillment of PhD
-- requirements at The Air Force Institute of Technology.
-----

```

```

library ieee;
use ieee.std_logic_1164.all;
use work.aim_math.all;
-----

```

```

-- ENTITY --
-----

```

```

entity adapt is
  generic (td : time := 1 ns;
          chan : natural := 10;
          rapid_decay : string := "rapid_decay.rom";
          compensate : string := "compensate.rom";
          K : natural := 5; -- integration filter constant *16
          L : natural := 11); -- integration filter constant *16
  port (din : in std_ulogic_vector(15 downto 0);
        clk : in std_ulogic;
        reset : in std_ulogic;
        drq : in std_ulogic;
        dxak : in std_ulogic;
        dout : out std_ulogic_vector(15 downto 0);
        dak : out std_ulogic;
        dxrq : out std_ulogic);
end adapt;

```

```

-- ARCHITECTURE --
-----

```

```

architecture str of adapt is

  component napfsm
    generic (td : time := 1 ns);
    port (ck : in std_ulogic; -- clock input
          drq : in std_ulogic; -- data request
          reset : in std_ulogic; -- reset
          dxak : in std_ulogic;
          neg : in std_ulogic;

          dak : out std_ulogic;

```

```

    lmp : out std_ulogic;
    s50 : out std_ulogic;
    s51 : out std_ulogic;
    s6  : out std_ulogic;
    s7  : out std_ulogic;
    s8  : out std_ulogic;
    s9  : out std_ulogic;
    s10 : out std_ulogic;
    mpy : out std_ulogic;
    lpf : out std_ulogic;
    adj : out std_ulogic;
    lr  : out std_ulogic;
    as  : out std_ulogic;
    lmic: out std_ulogic;
    pas : out std_ulogic;
    ltmp: out std_ulogic;
    wmc : out std_ulogic;
    wlp0 : out std_ulogic;
    wlp1 : out std_ulogic;
    zro : out std_ulogic;
    adv : out std_ulogic;
    dorq: out std_ulogic
);
end component;

component multiplier
    generic (w      : natural := 16;
             cont_td : time    := 1 ns;
             tdff_td : time    := 1 ns;
             addsub_td : time  := 1 ns;
             mux2_td  : time  := 1 ns;
             dffcl_td : time  := 1 ns;
             dff2_td  : time  := 1 ns);
    port (m1,m2      : in  std_ulogic_vector(w-1 downto 0);
          mlout      : out std_ulogic_vector(w-1 downto 0);
          c, cb      : in  std_ulogic;
          ld, ldb     : in  std_ulogic;
          result      : out std_ulogic_vector(2*w-1 downto 0));
end component;

component rom
    generic(tdr: time := 1 ns;          -- delay read
            w  : natural := 8;          -- width of each word
            nw : natural := chan;       -- number of words (rows)
            fn : string := "rom.data");
    port (addr : in  std_ulogic_vector(nw-1 downto 0);
          dout : out std_ulogic_vector(w-1 downto 0));
end component;

component ram
    generic(tdr, tdw : time := 1 ns; -- delay read, write
            w  : natural := 16;      -- width of each word
            nw : natural := chan;     -- number of words (rows)
            init : natural := 1024);  -- Initial values
    port (addr : in  std_ulogic_vector(nw-1 downto 0);
          din  : in  std_ulogic_vector(w-1 downto 0);

```

```

        wr : in std_ulogic;
        dout : out std_ulogic_vector(w-1 downto 0));
end component;

component row_sel
    generic (n : natural := chan; td : time := 1 ns);
    port(reset : in std_ulogic;
        adv : in std_ulogic;
        rows : out std_ulogic_vector(n-1 downto 0);
        rx : out std_ulogic
    );
end component;

component phi2
    generic (td : time := 1 ns);
    port (ck : in std_ulogic;
        ck1 : out std_ulogic;    -- in phase with ck
        ck2 : out std_ulogic);   -- out of phase with ck
end component;

component mux2
    generic (td : time := 1 ns;
        w : natural := 8 );
    port (i0, i1 : in std_ulogic_vector(w-1 downto 0);
        s : in std_ulogic;
        o1 : out std_ulogic_vector(w-1 downto 0));
end component;

component mux4
    generic (td : time := 1 ns;
        w : natural := 8 );
    port (i3, i2, i1, i0 : in std_ulogic_vector(w-1 downto 0);
        s1, s0 : in std_ulogic;
        o1 : out std_ulogic_vector(w-1 downto 0));
end component;

component alucell
    generic (td : time := 1 ns);
    port (a, b, ci, as, pass : in std_ulogic;
        sum, co : out std_ulogic);
end component;

component tnreg
    generic (td : time := 1 ns;    -- propagation delay
        n : natural := 8 );      -- number of bits
    port (din : in std_ulogic_vector(n-1 downto 0);
        dout : out std_ulogic_vector(n-1 downto 0);
        load : in std_ulogic;
        loadb : in std_ulogic);
end component;

component n_dffac1
    generic (n : natural := 8;
        td : time := 1 ns);
    port (d : in std_ulogic_vector(n-1 downto 0);
        c : in std_ulogic);

```



```

        reset: in std_ulogic;
        q,qb : out std_ulogic_vector(n-1 downto 0));
end component;

component nandgate
    generic (n : natural := 2;
            td : time := 1 ns);
    port (i : in std_ulogic_vector(n-1 downto 0);
          o : out std_ulogic);
end component;

component andgate
    generic (n : natural := 2;
            td : time := 1 ns);
    port (i : in std_ulogic_vector(n-1 downto 0);
          o : out std_ulogic);
end component;

component orgate
    generic (n : natural := 2;
            td : time := 1 ns);
    port (i : in std_ulogic_vector(n-1 downto 0);
          o : out std_ulogic);
end component;

component invert
    generic (td : time := 1 ns);
    port (i : in std_ulogic;
          o : out std_ulogic);
end component;

signal mpout          : std_ulogic_vector(35 downto 0);
signal rows           : std_ulogic_vector(chan-1 downto 0) := (others=>'0');
signal m5, m6         : std_ulogic_vector(17 downto 0);
signal riple         : std_ulogic_vector(17 downto 0);
signal rw0, rw1, rwi  : std_ulogic_vector(15 downto 0);
signal rw2           : std_ulogic_vector(15 downto 0);
signal m7,m8,ibus     : std_ulogic_vector(16 downto 0);
signal m9,m9_div2     : std_ulogic_vector(15 downto 0);
signal m5a, tmp       : std_ulogic_vector(15 downto 0);
signal adjout,lrout   : std_ulogic_vector(16 downto 0);
signal aluout, mic    : std_ulogic_vector(16 downto 0);
signal rw2in, ro2     : std_ulogic_vector(15 downto 0);
signal ro2b          : std_ulogic_vector(16 downto 0);
signal i61, i62, i63  : std_ulogic_vector(15 downto 0);
signal rol           : std_ulogic_vector( 7 downto 0);
signal a1, a2, a3, a4 : std_ulogic_vector( 1 downto 0);
signal a5, a6, o1, o2 : std_ulogic_vector( 1 downto 0);
signal lm1, lm2, mc1, mc2, lmpck, mp2ck : std_ulogic;
signal neg, rx, rxck, rxckr, lrck, lrckb : std_ulogic;
signal lmp, s50, s51, s6, s7, s8,s9,s10 : std_ulogic;
signal mpy, lpf, adj, lr, as, lmic, adv : std_ulogic;
signal pas, ltmp, wmc, zro, wlp0, wlp1 : std_ulogic;
signal lmic_ck, lmic_ckb, ltmpck       : std_ulogic;
signal xld1, xld2, zro_r, resetb       : std_ulogic;
signal gnd : std_ulogic := '0';

```

```

signal vdd : std_ulogic := '1';

begin

----- Finite State Machine
fsm : napfsm
  generic map(td=>td)
  port map(clk, drq, reset, dxak, neg, dak, lmp,
           s50, s51, s6, s7, s8, s9, s10, mpy,
           lpf, adj, lr, as, lmic, pas, ltmp, wmc,
           wlp0, wlp1, zro, adv, dxrq);

----- Memory address counter
rowsell1 : row_sel
  generic map(n=>chan, td=>1 ns)
  port map(rxckr, adv, rows, rx);

a1 <= rx & clk;
nand_1 : nandgate
  generic map(n=>2, td=>td)
  port map(a1, rxck);

inv_3 : invert
  generic map(td => td)
  port map(reset, resetb);

o1 <= rxck & resetb;
nand_2 : nandgate
  generic map(n=>2, td=>td)
  port map(o1, rxckr);

----- Read/Write Memories
ram0 : ram
  generic map( tdr=>td, tdw=>td, w=>16, nw=>chan, init=>0)
  port map(rows, m9, wlp0, rw0);

ram1 : ram
  generic map( tdr=>td, tdw=>td, w=>16, nw=>chan, init=>0)
  port map(rows, m9, wlp1, rw1);

m9_div2(14 downto 0) <= m9(15 downto 1);
m9_div2(15) <= '0';
rammux : mux2
  generic map(td=>td, w=>16)
  port map(m9, m9_div2, adj, rw2in);
ram2 : ram
  generic map( tdr=>td, tdw=>td, w=>16, nw=>chan, init=>1024)
  port map(rows, rw2in, wmc, rw2);

----- Read Only Memories
rom1 : rom
  generic map(tdr=>td, w=>8, nw=>chan, fn=>"compensate.rom")
  port map(rows, rol);

rom2 : rom
  generic map(tdr=>td, w=>11, nw=>chan, fn=>"rapid_decay.rom")

```

```

    port map(rows, ro2(10 downto 0));
    ro2(15 downto 11) <= (others=>vdd);    -- msbs are always '1'

----- Inversion for Rapid-Decay ROM
g1 : for i in 15 downto 5 generate
    inv_cell : invert
        generic map(td=>td)
        port map(ro2(i),ro2b(i-5));
end generate;
ro2b(16 downto 11)<=(others=>'0');

----- Multiplier Circuitry
mux10 : mux2
    generic map(td=>td, w=>16)
    port map(rw0, rw1, s10, rwi);

mux50 : mux2
    generic map(td=>td, w=>16)
    port map(m9, rwi, s50, m5a);

mux51 : mux2
    generic map(td=>td, w=>16)
    port map(m5a, rw2, s51, m5(15 downto 0));

m5(17 downto 16) <= (others=>gnd);    -- two extra bits for sign extension
                                     -- but values are always positive

i61( 7 downto 0) <= ro1;
i61(15 downto 8) <= (others=>gnd);
i62 <= i2uv(K,16);
i63 <= i2uv(L,16);
mux6 : mux4
    generic map(td=>td, w=>16)
    port map(i63,i62,ro2,i61, lpf,s6, m6(15 downto 0));
m6(17 downto 16) <= (others=>gnd);

mult1 : multiplier
    generic map(w=>18, cont_td=>td, tdff_td=>td, addsub_td=>td,
        mux2_td=>td, dffcl_td=>td, dff2_td=>td)
    port map(m5, m6, open, mc1, mc2, lm1, lm2, mpout);

phi2_1 : phi2
    generic map(td => td)
    port map(lmpck, lm1, lm2);

phi2_2 : phi2
    generic map(td => td)
    port map(mp2ck, mc1, mc2);

a2<= lmp & clk;
and_2 : andgate
    generic map(td=>td)
    port map(a2, lmpck);

a3<= mpy & clk;
and_3 : andgate
    generic map(td=>td)

```

```

    port map(a3, mp2ck);

----- Multiplier Output Mux (shift adjuster) and Latch
adj_mux : mux2
    generic map(td=>td, w=>17)
    port map(mpout(31 downto 15), mpout(25 downto 9), adj, adjout);

a4 <= clk & lr;
and_4 : andgate
    generic map(n=>2, td=>td)
    port map(a4, lrck);

inv_1 : invert
    generic map(td=>td)
    port map(lrck, lrckb);

mpylatch : tnreg
    generic map( td=>td, n=>17)
    port map(adjout, lrout, lrck, lrckb);

----- Data Input
phi2_3 : phi2
    generic map(td=>td)
    port map( drq, xld1, xld2);

tlatch1 : tnreg
    generic map( td=>td, n=>16)
    port map( din, ibus(16 downto 1), xld1, xld2);
ibus(0) <= '0'; -- no fraction

----- ALU and its Input Muxes
mux7 : mux2
    generic map(td=>td, w=>17)
    port map(lrout, ibus, s7, m7);

mux8 : mux2
    generic map(td=>td, w=>17)
    port map(ro2b, mic, s8, m8);

riple(0) <= as;
g2 : for i in 0 to 16 generate
    alu1 : alucell
        generic map(td =>td )
        port map( m7(i), m8(i), riple(i), as, pas, aluout(i), riple(i+1));
end generate;

----- ALU output latches
a5 <= lmic & clk;
and_5 : andgate
    generic map(n=>2, td=>td)
    port map(a5, lmic_ck);

mic_latch : n_dffa1
    generic map(n=>17, td=>td)
    port map( aluout, lmic_ck, reset, mic, open);

```

```

a6 <= ltmp & clk;
and_6 : andgate
    generic map(n=>2, td=>td)
    port map(a6, ltmpck);

o2 <= zro & reset;
or_2 : orgate
    generic map(n=>2, td=>td)
    port map(o2, zro_r);

tmp_dff : n_dffa1
    generic map( n => 16, td=>td)
    port map(aluout(16 downto 1), ltmpck, zro_r, tmp, open);

neg <= aluout(15);    -- don't latch, must know in state 6
dout <= tmp;          -- Final output

mux9 : mux2
    generic map(td=>td, w=>16)
    port map(mic(16 downto 1), tmp, s9, m9);

end str;

-----
--   CONFIGURATION   --
-----
configuration adapt_cfg of adapt is
for str
    for fsm : napfsm
        use configuration work.napfsm_cfg;
    end for;
    for row_sel1 : row_sel
        use configuration work.row_sel_cfg;
    end for;
    for all : andgate
        use entity work.andgate(beh);
    end for;
    for all : nandgate
        use entity work.nandgate(beh);
    end for;
    for all : orgate
        use entity work.orgate(beh);
    end for;
    for ram1, ram2 : ram
        use entity work.ram(beh);
    end for;
    for rom1, rom2 : rom
        use entity work.rom(beh);
    end for;
    for g1
        for all : invert
            use entity work.invert(beh);
        end for;
    end for;
    for all : mux2
        use entity work.mux2(beh);

```

```

end for;
for all : mux4
    use entity work.mux4(beh);
end for;
for all : phi2
    use configuration work.phi2_cfg;
end for;
for all : tnreg
    use configuration work.tnreg_cfg;
end for;
for g2
    for all : alucell
        use configuration work.alucell_cfg;
    end for;
end for;
for all : n_dffacl
    use configuration work.n_dffacl_cfg;
end for;
end for;
end adapt_cfg;

```

## Bibliography

1. R. D. Patterson, M. H. Allerhand, and C. Giguere, "Time-Domain Modelling of Peripheral Auditory Processing: A Modular Architecture and Software Platform," *Journal of the Acoustical Society of America*, vol. 98, pp. 1890–1894, 1995.
2. M. Lutnam and A. Martin, "Development of an Electroacoustic Analogue Model of the Middle Ear and Acoustic Reflex," *Journal of Sound and Vibration*, vol. 64, no. 1, pp. 133–157, 1979.
3. J. Zwislocki, "Analysis of the Middle-Ear Function. Part I: Input Impedance," *Journal of the Acoustical Society of America*, vol. 34, no. 8, pp. 1514–1523, 1962.
4. C. Giguere and P. C. Woodland, "A Computational Model of the Auditory Periphery for Speech and Hearing Research. I. Ascending Path," *Journal of the Acoustical Society of America*, vol. 95, no. 1, pp. 330–342, 1994.
5. R. Patterson and J. Holdsworth, "A Functional Model of Neural Activity Patterns and Auditory Images," in *Advances in Speech, Hearing and Language Processing* (W. Ainsworth, ed.), pp. Vol 3, Part B, JAI Press, London, 1996.
6. K. I. Francis and T. R. Anderson, "Binaural Phoneme Recognition Using the Auditory Image Model and Cross-Correlation," *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1998.
7. R. Patterson, T. R. Anderson, and M. Allerhand, "The Auditory Image Model as a Preprocessor for Spoken Language," in *Proc. Third ICSLP*, pp. 1395–1398, 1994.
8. J. Lazzaro and J. Wawrzynek, "Silicon Models for Auditory Scene Analysis," in *Advances in Neural Information Processing Systems 8* (M. Mozer, D. Touretsky, and M. Hasselmo, eds.), (Cambridge, MA), MIT Press, 1995.
9. R. F. Lyon, "CCD Correlators for Auditory Models," in *1991 Asilomar Conference on Signals, Systems and Computers*, 1991.
10. R. Patterson and J. Holdsworth, *An Introduction to Auditory Sensation Processing*. Cambridge CB2 2EF, England: MRC Applied Psychology Unit, 1990.
11. C. Giguere, *Speech Processing using a Wave Digital Filter Model of the Auditory Periphery*. PhD thesis, Department of Engineering and Darwin College, University of Cambridge, England, 1993.
12. M. Cooke, *Modelling Auditory Processing and Organisation*. Cambridge University Press, 1993.
13. E. de Boer and P. Kuyper, "Triggered Correlation," *IEEE Trans Bio Med Eng*, vol. BME15, no. 3, pp. 169–179, 1968.
14. P.I.M. Johannesma, "The Pre-response Stimulus Ensemble of Neurons in the Cochlear Nucleus," in *Symposium on Hearing Theory*, IPO, Eindhoven, 1972.

15. E. de Boer and H. de Jongh, "On Cochlear Encoding: Potentialities and Limitations of the Reverse-Correlation Technique," *Journal of the Acoustical Society of America*, vol. 63, pp. 115-135, 1978.
16. R. Patterson, I. Nimmo-Smith, J. Holdsworth, and P. Rice, "An Efficient Auditory Filterbank Based on the Gammatone Function," *Paper presented at a meeting of the IOC Speech Group on Auditory Modelling at RSRE*, 1987.
17. M. Slaney, "An Efficient Implementation of the Patterson-Holdsworth Auditory Filter Bank," Tech. Rep. 35, Apple Computer, Inc., Cupertino, CA 95014, 1993.
18. B. C. Moore and B. R. Glasberg, "Suggested Formulae for Calculating Auditory-filter Bandwidths and Excitation Patterns," *Journal of the Acoustical Society of America*, vol. 74, pp. 750-753, September 1983.
19. M. Slaney, "Lyons Cochlear Model," Tech. Rep. 13, Apple Computer, Inc., Cupertino, CA 95014, 1988.
20. B. Glasberg and B. Moore, "Derivation of Auditory Filter Shapes from Notched-noise Data," *Hearing Research*, vol. 47, pp. 103-108, 1990.
21. D. D. Greenwood, "A Cochlear Frequency-Position Function for Several Species 29 Years Later," *Journal of the Acoustical Society of America*, vol. 87, no. 6, pp. 2592-2605, 1990.
22. R. Patterson and B. Moore, "Auditory Filters and Excitation Patterns as Representations of Frequency Resolution," in *Frequency Selectivity in Hearing* (B. Moore, ed.), pp. 123-177, Academic, London, 1986.
23. "AIM Documentation." On-line documentation file shipped with AIM release 8. April 1997.
24. R. Meddis, "Simulation of Auditory-Neural Transduction: Further Studies," *Journal of the Acoustical Society of America*, vol. 83, pp. 1056-1063, 1988.
25. K. I. Francis, *Speaker Independent Phoneme Recognition with a Binaural Auditory Image Model*. PhD thesis, Department of Electrical Engineering, University of Dayton, Dayton, Ohio, 1997.
26. R. Patterson, J. Holdsworth, and M. Allerhand, "Auditory Models as Preprocessors for Speech Recognition," in *The Auditory Processing of Speech: From the Auditory Periphery to Words* (M. Schouten, ed.), pp. 67-83, Mouton de Gruyter, 1992.
27. S. SanGregory, C. Brothers, D. Gallager, and R. Siferd, "A Fast Low-Power Logarithm Approximation Technique and Circuit," in *Proceedings of the 42<sup>nd</sup> Midwest Symposium on Circuits and Systems*, IEEE Press, New York, August 1999.
28. J. N. Mitchell, "Computer Multiplication and Division Using Binary Logarithms," *IRE Transactions on Electronic Computers*, pp. 512-517, August 1962.
29. M. Combet, H. Zonneveld, and L. Verbeek, "Computation of the Base Two Logarithm of Binary Numbers," *IEEE Trans Electronic Computers*, pp. 863-867, December 1965.
30. E. L. Hall, D. D. Lynch, and S. J. Dwyer III, "Generation of Products and Quotients Using Approximate Binary Logarithms for Digital Filtering Applications," *IEEE Transactions on Computers*, vol. C-19, pp. 97-105, February 1970.



31. A. Brown, ed., *VLSI-Circuits and Systems in Silicon*. McGraw-Hill, 1991.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE December 1999		3. REPORT TYPE AND DATES COVERED Ph.D. Dissertation
4. TITLE AND SUBTITLE Approximation and Optimization of an Auditory Model for Realization in VLSI Hardware			5. FUNDING NUMBERS	
6. AUTHOR(S) Samuel L SanGregory				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583			8. PERFORMING ORGANIZATION REPORT NUMBER  AFIT/DS/ENG/99-07	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dr. Timothy Anderson Air Force Research Laboratories (AFRL/HECA) Wright-Patterson Air Force Base, OH 45433 (937) 255-8914			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION AVAILABILITY STATEMENT			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) <p>The Auditory Image Model (AIM) is a software suite developed to functionally model the role of the ear in the human hearing process. AIM includes detailed filter equations for the major functional portions of the ear. Currently, AIM is run on a computer workstation and requires 10 to 100 times real-time to process audio information and produce an auditory image.</p> <p>An all digital approximation of AIM which is suitable for implementation in very large scale integrated circuits (VLSI) is presented. This document details the mathematical models of AIM, the approximations and optimizations used to simplify the model, and an efficient multi-rate architecture designed to execute the new model. Simulations indicate that the VLSI architecture can sustain real-time operation for a 32 filter channel system. Simulations also indicate that the resulting chip will consume an estimated 18 mW of power and fit inside a die which is 3 mm square.</p> <p>Additionally, the details of a new, and efficient method to compute an approximate logarithm (base two) on binary integers is included. The approximate logarithm algorithm is used to convert sound energy into millibells quickly and with little power. Additionally, the algorithm is easily extended to compute an approximate logarithm in base ten which broadens the class of problems to which it may be applied.</p>				
14. SUBJECT TERMS auditory model, phoneme recognition, speech, hearing, digital filter, logarithm, VLSI, VHDL, digital signal processing, spectral analysis, neural activity pattern			15. NUMBER OF PAGES 153	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	